XL C/C++ Advanced Edition for Mac OS X

Getting Started with XL C/C++ for Mac OS X

Version 6.0

SC09-7859-00

XL C/C++ Advanced Edition for Mac OS X

IBM

Getting Started with XL C/C++ for Mac OS X

Version 6.0

SC09-7859-00

Note:
Before using this information and the product it supports, be sure to read the information in "Notices" on page 61.

First Edition (December, 2003)

This edition applies to Version 6, Release 0, Modification 0 of XL C/C++ Advanced Edition Version 6.0 for Mac OS X (product number 5724–G12) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them by the Internet to the following address:

`compinfo@ca.ibm.com`

Include the title and order number of this book, and the page number or topic related to your comment. Be sure to include your e-mail address if you want a reply.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this book

XL C/C++ Advanced Edition Version 6.0 for Mac OS X is an optimizing, standards-based, command-line compiler for the Mac OS X operating system running on the PowerPC® architecture. The compiler is a professional programming tool for creating and maintaining 32-bit applications in the extended C and C++ programming languages.

This book introduces you to the XL C/C++ for Mac OS X compiler. It contains instructions for installing and setting up the compilation environment. Testing the installation by compiling a Hello World application provides an introduction to invoking the compiler and controlling the compilation process through compiler options. This book also describes the types of transformations the compiler can perform, the accepted file types for input and output, and general advice for porting an existing application. This book also provides an introduction to optimizing the performance of your applications. It delineates ways to exercise the capabilities of the compiler to exploit the multilayered architecture of the PowerPC processor.

If you are porting an application previously developed with IBM C for AIX, VisualAge C++ Professional for AIX, or VisualAge C++ for Linux, your makefiles can be adapted to work on the Mac OS X platform. If you are new to the XL C/C++ compiler, this book can open a path to improved performance during compile, link, and run time.

This document assumes that you are familiar with the C and C++ programming languages, the Mac OS X operating system, the TENEX C shell (*tcsh*), and the Bourne-Again shell (*bash*).

Highlighting conventions

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
Italics	Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.
`Example`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Examples are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of the language constructs. Some examples are only code fragments and will not compile without additional code.

How to read the syntax diagrams

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶─── symbol indicates the beginning of a command, directive, or statement.

Reading the Syntax Diagrams

The ───► symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ►─── symbol indicates that a command, directive, or statement is continued from the previous line.

The ───►◄ symbol indicates the end of a command, directive, or statement.

Diagrams of syntactical units other than complete commands, directives, or statements start with the ►─── symbol and end with the ───► symbol.

Note: In the following diagrams, `statement` represents a C or C++ C command, directive, or statement.

- Required items appear on the horizontal line (the main path).

```
►►──statement──required_item──────────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──statement──┬───────────────┬──────────────────────────────────────────►◄
               └─optional_item─┘
```

- If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──statement──┬─required_choice1─┬───────────────────────────────────────►◄
               └─required_choice2─┘
```

  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──statement──┬──────────────────┬───────────────────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```

  The item that is the default appears above the main path.

```
               ┌─default_item───┐
►►──statement──┴─alternate_item─┴─────────────────────────────────────────►◄
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
               ┌─────────────────┐
               ▼                 │
►►──statement────repeatable_item─┴────────────────────────────────────────►◄
```

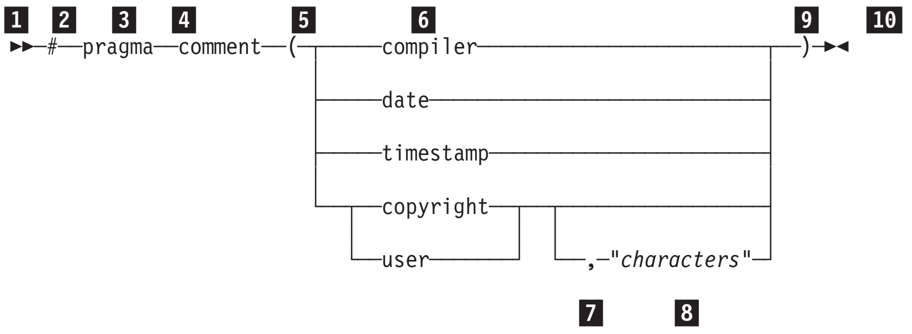
  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

  Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

The following syntax diagram example shows the syntax for the **#pragma comment** directive. See *XL C/C++ for Mac OS X C/C++ Language Reference* for information on the **#pragma** directive.



1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword `pragma` must appear following the # symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

XL C/C++ for Mac OS X Overview

XL C/C++ Advanced Edition Version 6.0 for Mac OS X is an optimizing, standards-based, command-line compiler for the Mac OS X operating system, running on PowerPC hardware with the PowerPC architecture. XL C/C++ for Mac OS X uses the Mac OS X, GNU C, and GNU C++ header files and the GNU C and C++ run-time libraries to produce code that is binary compatible with that produced with the GNU compiler, gcc Version 3.3. Portions of an application can be built with XL C/C++ for Mac OS X and combined with portions built with gcc to produce an application that behaves as if it had been built solely with gcc. To ensure that the proper versions of headers and run-time libraries are present on the system, the gcc 3.3 compiler must be installed before installing XL C/C++ Advanced Edition Version 6.0 for Mac OS X.

The compiler enables application developers to create and maintain optimized 32-bit applications for the Mac OS X platform. The compiler brings to the Mac OS X platform the flexibility, features, and refinements that evolved from its releases for the AIX operating system and the Linux platform. The compiler accommodates a wide range of program subtleties, often without requiring significant amounts of rework. These advantages are now available to Macintosh application developers because of the underlying UNIX operating system of the Mac OS X platform. With the addition of language extensions to support the Motorola AltiVec Programming Interface and the Apple Velocity Engine, the compiler offers a diversified portfolio of optimization techniques tailored for the PowerPC 970 architecture.

XL C/C++ Advanced Edition Version 6.0 for Mac OS X can be used for large, complex, and computationally intensive programs. It supports interlanguage calls with XL Fortran. For applications that require SIMD (single-instruction, multiple data) parallel processing, performance improvements can be achieved through traditional optimization techniques, which may be less labor-intensive than vector programming. Many of the optimizations developed by IBM are controlled by compiler options, pragmas, or compiler invocation mode.

Source compatibility

The compiler supports the two ISO programming language specifications for C: ISO/IEC 9899:1990 (referred to as C89) and ISO/IEC 9899:1999 (C99). Certain features specified in the C99 Standard require corresponding run-time library support, which may not be available in the current operating system release.

In addition, the compiler implements extensions to the C language, which include a subset of the GNU C language extensions and the language extensions necessary to support AltiVec vector programming.

The compiler supports Standard C++, which is the programming language described in the international standard ISO/IEC 14882:1998(E) and based on C89. The compiler also implements extensions to Standard C++ for compatibility with XL C and selected features of C99 and g++.

Command line compatibility

XL C/C++ for Mac OS X supports a subset of the GNU compiler command options to facilitate porting applications developed with `gcc` and `g++`. This support is available when the gxlc or gxlc++ invocation command is used.

Coexistence with open source resources

To achieve binary compatibility with gcc-compiled code, a program compiled with XL C/C++ for Mac OS X includes the same headers as those used by a GNU compiler residing on the same system. The compiler optimizes the program while maintaining binary compatibility with objects produced by gcc Version 3.3. Some noteworthy points about this relationship are:

- IBM built-in functions for PowerPC processors coexist with GNU C built-ins.
- Compilation of C and C++ programs uses the GNU C, GNU C++, and Mac OS X header files.
- Compilation uses the GNU assembler for assembler input files.
- Linking uses the Mac OS X linker.
- Compiled code is linked to the GNU C and Mac OS X run-time libraries.
- Compiled C++ code is linked to the GNU C, GNU C++, and Mac OS X run-time libraries.
- Debugging uses the GNU debugger, gdb.

National Language Support

The compiler can be used to create and maintain applications that require multibyte characters or Unicode support.

Libraries

The compiler uses the GNU C and C++ headers, and the resulting application is linked with the C and C++ run-time libraries provided with gcc Version 3.3. IBM implementations of some C header files, such as `stdarg.h` and `varargs.h`, ship with the product and override the corresponding gcc header files. These implementations are functionally equivalent to the corresponding GNU C implementations. Other IBM headers are wrappers that include some gcc header files.

The header files `wchar.h`, `wctype.h`, and `iso646.h` may not be available at certain release levels of the C run-time library provided with gcc. XL C/C++ provides versions of these header files, which are used if they are not otherwise present on the system. In this case, specifying the macro -D_XLC_WC_H with the compiler invocation will suppress the diagnostic messages related to the use of the XL C/C++ versions. However, the header files provided by the system C run-time library should always be used if they are available.

Utilities and commands

XL C/C++ Advanced Edition Version 6.0 for Mac OS X provides the following specialized commands to aid program development. For more information, refer to *XL C/C++ for Mac OS X Compiler Reference*.

vacpp_configure Utility
A program that creates the configuration file vac.cfg, which specifies the location of the GNU compiler and other configuration information. The IBM C and C++ compilers both use vac.cfg for their configuration.

cleanpdf Command
A command related to profile-directed feedback used for managing the PDFDIR directory. Removes all profiling information from the specified, PDFDIR, or current directory.

resetpdf Command
Current behavior is the same as the **cleanpdf** command. The command is retained for compatibility with earlier releases on other platforms.

gxlc and gxlc++ Utilities
Invocation methods that translate a GNU C or GNU C++ invocation command into a corresponding **xlc** or **xlc++** command and invoke the XL C/C++ compiler. The purpose of these utilities is to minimize the number of changes to makefiles used for existing applications built with the GNU compilers and to facilitate the transition to XL C/C++.

Documentation and online help

XL C/C++ Advanced Edition Version 6.0 for Mac OS X provides product documentation in the following formats:
- Readme files
- Man pages
- HTML files
- PDF documents

By default, the PDF documents are installed in the /opt/ibmcmp/vacpp/6.0/doc/en_US/pdf directory.

In addition to this guide, XL C/C++ for Mac OS X provides the following PDF documents:

XL C/C++ for Mac OS X C/C++ Language Reference **(language.pdf)**
Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to non-proprietary standards.

XL C/C++ for Mac OS X Compiler Reference **(compiler.pdf)**
Contains information about the various compiler options, pragmas, macros, and built-in functions, including those used for parallel processing.

XL C/C++ for Mac OS X Programming Tasks **(proguide.pdf)**
Contains information about programming using XL C/C++ for Mac OS X not covered in other publications.

The HTML version of the documentation is searchable as part of the Apple Help Center. To display the HTML files, select **Help** from the Task bar. Then select **Mac Help > IBM XL C/C++ Compiler**.

Installing XL C/C++ for Mac OS X

XL C/C++ for Mac OS X is a command-line application that installs from CD-ROM or from a downloadable disk image (.dmg). The installation procedures are the same for single- or multiuser environments. Both XL C and C++ compilers are contained in the metapackage vacpp.cmp.mpkg, which can be installed through the GUI or from the command line, using the Mac OS X Installer.app utility. Prior to installation, you can get more information on the metapackage by issuing the following command:

```
/usr/sbin/installer -pkg vacpp.cmp.mpkg -pkginfo -verbose
```

Installation from the GUI allows you to install the compiler in any directory. A command-line installation installs the C compiler in the default location, the /opt/ibmcmp/vac/6.0 directory, and the C++ compiler in the default location, the /opt/ibmcmp/vacpp/6.0 directory. The directory where the compiler is installed is referred to as the *root path* or *base relocation*.

XL C/C++ for Mac OS X uses the GNU C and C++ header files and run-time libraries to produce code that is binary compatible with that produced with the GNU compiler, gcc Version 3.3. To ensure that the proper versions of headers and run-time libraries are present on the system, gcc 3.3 must be installed before installing XL C/C++ for Mac OS X.

XL C/C++ for Mac OS X must therefore be configured so that it can find a GNU compiler, header files, and libraries (including crt.o files), regardless of its base relocation. The installation program attempts to create the default configuration file `/etc/opt/ibmcmp/vac/6.0/vac.cfg` automatically if all prerequisites have been installed. The initial configuration can also be handled by manually running one of the related convenience tools: the new_install command or the vacpp_configure utility. Both have the ability to create the default configuration file.

The default configuration file can subsequently be modified by rerunning the vacpp_configure utility with different options. Alternatively, for temporary changes to the vacpp_configure settings, you can override each setting through a command-line option, or you can use a different configuration file by compiling with the **-F** option and specifying a particular configuration file as the argument to the option. See *XL C/C++ for Mac OS X Compiler Reference* for more information.

System requirements

Operating system
Mac OS X, Version 10.2 or 10.3

Hardware
Power Mac G4 or G5

The compiler, its generated object programs, and run-time library run on Apple Power Mac G4 or G5 systems with the required software and disk space. To take maximum advantage of different hardware configurations, the compiler provides a number of options for performance tuning based on the configuration of the machine used for executing an application. However, the compiler Scheduler models only the G5 architecture and does not optimally exploit the G4 architecture.

Disk space

At least 150 MB for product packages.

At least 200 MB (recommended) to accommodate high levels of optimization, which may require significant amounts of additional paging and temporary disk space.

Required software

The Mac OS X Developer Tools package, which includes gcc Version 3.3. The package is available from the Apple Developer Connection Member Site at http://connect.apple.com/ . For Mac OS X 10.2, the upgrade to gcc Version 3.3 requires installation of the Apple package August2003gccUpdater.pkg. For Mac OS X 10.3 (Panther), Xcode Tools v1.0 or later is required.

Prerequisite tasks and conditions

The compiler requires at least one instance of gcc Version 3.3 on the system. You will need to provide its location if you have to manually configure XL C/C++ after installation.

A installation of all packages requires at least 150 MB of disk space. However, compiling at higher levels of optimization may require large amounts of temporary disk space. The recommended amount of temporary disk space is a minimum of 200 MB.

You must remove any beta version of the C or C++ compiler from your system.

The prerequisite tasks for a first-time installation are:

1. Upgrade to gcc Version 3.3 from the Mac OS X Developer Tools CD-ROM or from the Apple Developer Connection Member Site at http://connect.apple.com. For Mac OS X 10.2, the upgrade to gcc Version 3.3 requires installation of the Apple package August2003gccUpdater.pkg. For Mac OS X 10.3 (Panther), Xcode Tools v1.0 or later is required.

   XL C/C++ for Mac OS X does not require the gcc executable to be in a particular location. When you configure XL C/C++ for Mac OS X before first use, you save the location of the gcc compiler on which XL C/C++ for Mac OS X relies, in the XL C/C++ configuration file.
2. Verify that you have sufficient space. The following command queries the volumes into which you intend to install the compiler package.

   ```
   /usr/sbin/installer -pkg absolute_path/vacpp.cmp.mpkg -volinfo -verbose
   ```

   where *absolute_path* is the absolute path to the metapackage.

Installation procedures for XL C/C++ for Mac OS X

The GUI installation uses Installer.app, an application that installs with the operating system. A GUI installation allows you to specify both the volume and base relocation for the compilers. The installation program enables permissions on the relocation directory for all users. A command-line installation allows you to specify only the volume, and uses the default installation locations.

The high-level steps are the same for either type of installation. These steps are as follows:

1. Become the root user or a user with administrator privileges.
2. Mount the destination volume for the compiler installation.

3. Install the packages.
4. Configure the compiler. This means checking for the existence of the default configuration file for the compiler. If none was created automatically by the installation program, create it by running either the new_install command or the vacpp_configure utility.
5. Ensure that the NLSPATH includes search paths for any relocated run-time message catalogs.
6. Test the installation.

The following table lists the packages and their prerequisites. The packages in the Prerequisites column are presented in the order in which they must be installed.

XL C++ packages in **vacpp.cmp.mpkg**

Package name	**Prerequisites**	**Description**
xlsmp.msg.rte.pkg	none	SMP run-time messages
xlsmp.rte.pkg	xlsmp.msg.rte.pkg	SMP run-time dynamic libraries
xlsmp.lib.pkg	xlsmp.msg.rte.pkg xlsmp.rte.pkg	SMP run-time static libraries
vac.lic.pkg	none	C compiler license
vac.cmp.pkg	gcc 3.3 vac.lic.pkg xlsmp.msg.rte.pkg xlsmp.rte.pkg xlsmp.lib.pkg	The C compiler
vac.samples.pkg	none	Example code for the C compiler
vacpp.rte.pkg	none	C++ run-time libraries
vacpp.cmp.pkg	vac.cmp.pkg	The C++ compiler
vacpp.help.pkg	none	C++ compiler documentation
vacpp.samples.pkg	none	Example code for the C++ compiler

Relocatable packages

You can relocate the packages `xlsmp.msg.rte.pkg`, `xlsmp.rte.pkg`, and `xlsmp.lib.pkg` to a directory different from the default installation location. However, you must use the same directory for all three packages.

Similarly, you can relocate the compiler packages `vac.cmp.pkg` and `vacpp.cmp.pkg` to a directory different from the original installation location. You must use the same directory for both packages. However, the package for the C++ run-time environment, `vacpp.rte.pkg`, can be relocated independently.

The packages `vac.lic.pkg`, `vac.samples.pkg`, `vacpp.samples.pkg`, and `vacpp.help.pkg` are independent and can be relocated to any directory or directories. However, installing `vac.lic.pkg` is mandatory, whereas installing `vac.samples.pkg`, `vacpp.samples.pkg`, or `vacpp.help.pkg` is optional.

Installing through the GUI

A GUI installation allows you to specify both the destination volume and a nondefault relocation.

1. Become the root user or a user with administrator privileges.

2. Mount the CD-ROM device.
3. If necessary, mount the destination volume for the compiler installation.
4. Locate the file `vacpp.cmp.mpkg`, which is in the `packages` directory of the CD-ROM or disk image.
5. Double-click the metapackage icon.
6. If an Authenticate window appears, enter the root user ID and password, or the user ID and password of a user with administrator privileges.
7. A message tells you that this Installer package needs to run a program to determine if it can be installed. Click **Continue**.
8. An Introduction page appears. After reading it, click **Continue**.
9. A Read Me page appears. After reading it, click **Continue**.
10. The License page appears. After reading it, click **Continue**.
11. Click **Agree** to proceed with the installation.
12. A Select Destination page appears. Specify a destination volume.
13. You can specify the installation location.

    By default, XL C/C++ installs the C compiler in the following directory on the specified volume:

    ```
    /opt/ibmcmp/vac/6.0
    ```

    The C++ compiler installs in:

    ```
    /opt/ibmcmp/vacpp/6.0
    ```

    To accept the default relocation, click **Continue**.

    To install the product in a different directory, click **Choose...**. Navigate to a directory (*relocation_path*). On the specified volume, XL C/C++ installs in the following directories:

    ```
    relocation_path/opt/ibmcmp/vac/6.0
    relocation_path/opt/ibmcmp/vacpp/6.0
    ```

    The specified location will be referred to as your *base relocation*. Both compilers must use the same base relocation.
14. An Installation Type page appears. To install all packages, click **Install**.

    To install the XL C/C++ packages selectively, click **Customize**.

    Deselect the packages you do not want to install at this time, heeding any package prerequisites. Click **Install**.
15. A notification message appears, indicating that the selected packages installed successfully. Check to see if the installation program created the default configuration file /etc/opt/ibmcmp/vac/6.0/vac.cfg. To do this, select **File > Show Log** from the Installer menu bar. If the installation program failed to create the configuration file, you will need to install any missing prerequisites and possibly configure the compiler manually by running new_install or vacpp_configure.
16. Click **Close** to exit Installer.

Command-line installation

A command-line installation allows you to specify only the destination volume. The C compiler installs in the default location /opt/ibmcmp/vac/6.0. The C++ compiler installs in /opt/ibmcmp/vacpp/6.0.

To install XL C/C++ from the command line, do the following:

1. Become the root user or a user with administrator privileges.
2. Mount the destination volume for the compiler installation.
3. Issue the following command:

   ```
   /usr/sbin/installer -pkg absolute_path/vacpp.cmp.mpkg -target / -dumplog
   ```

   where `absolute_path` is the absolute path to the metapackage. If you are installing from CD-ROM, the path is `/Volumes/CDROM/packages/`. The `-dumplog` option writes installation information to the screen.
4. Check the log information to see if the installation program created the default configuration file /etc/opt/ibmcmp/vac/6.0/vac.cfg. If it did not, you will need to install any missing prerequisites and possibly configure the compiler manually by running vacpp_configure.

Enabling the compiler man pages

To enable the XL C/C++ man pages for the compiler invocation commands and other command-line utilities, you must add the XL C/C++ man directory to the beginning of the MANPATH environment variable. You do not need to rebuild the `catman` database for the XL C/C++ man pages to appear in a `man -k` (`whatis`) or `apropos` command.

To enable the XL C/C++ man pages, do the following:

1. Check the current value of the MANPATH environment variable:

   ```
   echo $MANPATH
   ```
2. If `echo $MANPATH` does not return a value, set the MANPATH environment variable to the following:

   ```
   /opt/ibmcmp/vacpp/6.0/man/en_US/man:/usr/share/man
   ```

   or, if you used a relocation,

   ```
   base_relocation/opt/ibmcmp/vacpp/6.0/man/en_US/man:/usr/share/man
   ```

   The system-default man pages use the search path `/usr/share/man`, which is, by default, hidden.

   Otherwise, if `echo $MANPATH` returns a value, add the path to the XL C/C++ man directory to the beginning of the MANPATH.

   From the tcsh shell, the command is:

   ```
   setenv MANPATH /opt/ibmcmp/vacpp/6.0/man/en_US/man:$MANPATH
   ```

   or, if you used a relocation,

   ```
   setenv MANPATH base_relocation/opt/ibmcmp/vacpp/6.0/man/en_US/man:$MANPATH
   ```

   From the bash shell, the command is:

   ```
   export MANPATH=/opt/ibmcmp/vacpp/6.0/man/en_US/man:$MANPATH
   ```

   or, if you used a relocation,

   ```
   export MANPATH=base_relocation/opt/ibmcmp/vacpp/6.0/man/en_US/man:$MANPATH
   ```
3. Test the environment by executing the **man** command on one of the compiler invocation commands. For example, the command

   ```
   man xlc++
   ```

   should return the XL C/C++ man page.

The command

```
man -a cc
```

should return all man pages present on the system for the **cc** command.

Viewing the product documentation

In addition to the man pages, XL C/C++ for Mac OS X provides softcopy documentation that installs as part of the Help package (vacpp.help.pkg). Upon successful installation, the softcopy documentation is accessible and searchable through the Finder Help menu.

The product documentation is provided in the following formats:

Readme file	A readme file is located at the root directory of the installation CD. The file installs in the `*base_relocation*/opt/ibmcmp/vacpp/6.0/doc/en_US` directory.
PDF books	The PDF version of the XL C/C++ documentation is located in the `/doc/en_US/pdf` directory of the installation CD. The PDF files install in the *base_relocation*`/opt/ibmcmp/vacpp/6.0/doc/en_US/pdf` directory.
HTML files	The HTML version of the XL C/C++ documentation installs in the *base_relocation*`/opt/ibmcmp/vacpp/6.0/doc/en_US/html` directory. You can display the HTML from this directory or through the Finder Help menu.
Man pages	Man pages for the compiler invocation commands and command-line utilities that ship with this product are located in the *base_relocation*`/opt/ibmcmp/vacpp/6.0/man/en_US/man` directory.

To access the searchable XL C/C++ documentation, select **Help > Mac Help > IBM XL C/C++ Compiler** from the Finder menu bar.

Configure the compiler

A compiler configuration file stores settings such as the location of the compiler and the install paths to the gcc binary, headers, and libraries. Before you can use the compiler, you must have a default configuration file. The default configuration file provides the configuration for the compiler that is used if no other configuration file is specified at compile time. The fully qualified name of the default configuration file is `/etc/opt/ibmcmp/vac/6.0/vac.cfg`, regardless of any relocations you might have used. Both XL C/C++ compilers use this configuration file.

The installation program attempts to create the default configuration file when it completes installing all prerequisite packages for the compiler. If a default configuration file already exists, the installation program creates a new one. The previous configuration file is backed up, and a timestamp is appended to the file name.

The success or failure of the installation program to configure the compiler is an issue separate from the installation of the software packages themselves. The installation program emits diagnostic messages related to its attempts to configure the compiler. To check whether the default configuration file was created, select **File > Show Log** before exiting the Installer program. If you installed packages from the command-line without using the **-dumplog** option, you will need to check for the existence of the file `/etc/opt/ibmcmp/vac/6.0/vac.cfg`.

If you installed all required packages but did not get a default configuration file, you must run either the vacpp_configure utility or the new_install command, which are utilities that facilitate creating a compiler configuration file. The vacpp_configure utility helps you create a properly formed configuration file. The new_install command queries the /Library/Receipts directory. It uses the results of its queries to construct a vacpp_configure command that creates the default configuration file for your system; new_install then invokes vacpp_configure.

The new_install command is intended to be used only for the first configuration of the compiler prior to first use; vacpp_configure should be used for reconfiguration.

Note: You must run vacpp_configure if the gcc compiler on your system is changed or moved.

To run new_install, do the following:

Step 1. Change directories so that your working directory is /opt/ibmcmp/vacpp/6.0/bin if you used the default installation location, or *relocation*/opt/ibmcmp/vacpp/6.0/bin if you used a relocation.

Step 2. Execute the command:

```
./new_install
```

which is equivalent to the vacpp_configure command:

```
relocation/opt/ibmcmp/vacpp/6.0/bin/vacpp_configure
   -install
   -gcc gccpath
   -smprt relocation/opt/ibmcmp/xlsmp/1.4
   -vac relocation/opt/ibmcmp/vac/6.0
   -vacpprt relocation/opt/ibmcmp/vacpp/6.0
   -vacpp relocation/opt/ibmcmp/vacpp/6.0
   -vaclic license_relocation/opt/ibmcmp/vac/6.0/lib/libxlcmp.dylib
   relocation/opt/ibmcmp/vac/6.0/etc/vac.base.cfg
```

If new_install exits with an error, it means that all the required information could not be determined automatically. You will need to run the vacpp_configure utility manually.

Related References
- “The vacpp_configure utility” on page 18
- “Customizing the compilation environment” on page 17

Setting the correct NLSPATH

If you installed the run-time libraries for the compiler or SMP to non-default locations, you must set the NLSPATH to reflect the new locations for the run-time message catalogs. An incorrect NLSPATH does not affect compilation. However, when you run the program without having modified the NLSPATH, the message catalogs might not be found.

Issue the following commands to set up NLSPATH for a relocated installation:

```
export NLSPATH=$NLSPATH:smprt_path/opt/ibmcmp/msg/en_US/%N
                      :relocation_path/opt/ibmcmp/vacpp/6.0/msg/en_US/%N
```

Note: The command is a single line. The line breaks shown are for legibility and formatting on the printed page.

Uninstalling XL C/C++ for Mac OS X

You must have root user or administrator access to uninstall the product.

Some packages should not be uninstalled if IBM XL Fortran is installed on the same system and you want to keep it on that system. XL Fortran requires the following packages for the SMP run time: `xlsmp.rte`, `xlsmp.msg.rte` and `xlsmp.lib`. Applications built with the XL C, C++, and Fortran compilers might have been linked to these run-time libraries. If so, they will stop functioning if these run-time libraries are removed.

However, if you are uninstalling all IBM compilers from your system, you can safely delete these files and the others that are shared by the XL C, C++, and Fortran compilers: `/opt/ibmcmp/lib`, `/opt/ibmcmp/msg`, and `/opt/ibmcmp/xlsmp`.

1. Issue the following command to remove the Library Receipts for the IBM C and C++ compilers, samples, and documentation:

   /bin/rm -rf /Library/Receipts/vac*.pkg
2. Remove the default configuration file for the compiler and the configuration file for gxlc:

   /bin/rm /etc/opt/ibmcmp/vac/6.0/vac.cfg

   /bin/rm /etc/opt/ibmcmp/vac/6.0/gxlc.cfg
3. Remove the compiler license:

   /bin/rm -f *relocation***/opt/ibmcmp/vac/6.0/lib/libxlcmp.dylib**
4. Remove the XL C/C++ compiler from the Xcode IDE by running the following script:

   *relocation***/opt/ibmcmp/vacpp/6.0/exe/xlc_ide_plugin_uninstall**
5. Remove the compiler and its documentation:

   /bin/rm -rf *relocation***/opt/ibmcmp/vac**

   /bin/rm -rf *relocation***/opt/ibmcmp/vacpp**
6. Remove the symbolic link to the XL C/C++ for Mac OS X help:

   /bin/rm /Library/Documentation/Help/ibmxlcpp
7. If you no longer need the SMP libraries, which are used by both XL Fortran and XL C/C++, delete them as follows. Do not delete these libraries if you have XL C/C++ installed on your system or if you have any Fortran, C, or C++ applications that link to these libraries.

   /bin/rm -rf /Library/Receipts/xlsmp.*.pkg

   /bin/rm -rf /opt/ibmcmp/lib/libxlomp_ser.*.dylib

   /bin/rm -rf /opt/ibmcmp/lib/libxlsmp.*.dylib

   /bin/rm -rf /opt/ibmcmp/msg/en_US/smprt.cat

   /bin/rm -rf /opt/ibmcmp/xlsmp
8. If you no longer need the C++ run-time libraries, vacpp.rte.pkg, delete them as follows. Do not delete these libraries if you have any C++ applications built with XL C/C++.

   /bin/rm -rf /Library/Receipts/vacpp.rte.pkg

   /bin/rm -rf /opt/ibmcmp/lib/libibmc++.*.dylib

Testing the installation

To test the product installation and the critical search paths, try compiling simple applications in C phase by phase. For convenience, you might want to add the path to the XL C/C++ executable to your PATH environment variable.

Related References
- “Customizing the compilation environment” on page 17

Building *Hello World* in C and C++

Test the installation and configuration by creating classic Hello World programs in C and C++ and compiling each in successive phases. The C test verifies whether the configuration finds the GNU C compiler, the GNU C headers, and the GNU C run-time libraries. The second C test verifies that the language extensions for AltiVec vector programming are accepted and properly handled by the compiler. The C++ test confirms that the C++ compiler, the GNU C++ run-time library and the C++ headers can be found.

Test the C compiler

1. In a text editor, create a C program similar to the following listing, and name the source file `hello.c`.

   ```
   #include <stdio.h>
   int main(void)
   {
      printf("Hello World!\n");
      return 0;
   }
   ```
2. Compile step. Run the command

   */base_relocation***/opt/ibmcmp/vac/6.0/bin/xlc -c hello.c -o hello.o**, which emits no diagnostic messages. Running a variation of this command emits all pertinent IBM diagnostic messages, **xlc -c -qinfo=all hello.c -o hello.o**.

   Note: ▶ C The compiler option **-qinfo** is equivalent to **-qinfo=all**.

   In these commands, the name of the output file, `hello.o`, is the same as the default name assigned by the compiler. When the name you want for the output file is the same as that assigned by the compiler, the name of the output file is optional in the command.

   The variation is analogous to the GNU C command

   ```
   gcc -c -Wall -D_GNU_SOURCE hello.c -o hello.o
   ```

   The IBM compiler also defines the macro `_GNU_SOURCE` to the value 1 and emits all levels of diagnostic messages.
3. Link step. From the directory into which you installed XL C/C++, run the command

   */base_relocation***/opt/ibmcmp/vac/6.0/bin/xlc -o hello hello.o**

   This command is analogous to the GNU C command

   ```
   gcc hello.o -o hello
   ```
4. Run the program:

   ./hello

   The expected result is to display `"Hello World!"` to the screen.
5. Check the exit code of the program:

   echo $?

The return value should be zero.

Test the C++ compiler

Repeat the test by creating a C++ ″Hello World″ program and compiling each phase separately.

1. In a text editor, create the ″Hello World″ program in C++, and name the source file `hello.C`.

   ```
   #include <iostream>
   int main()
   {
      std::cout << "Hello World!" << std::endl;
      return 0;
   }
   ```
2. Compile step. Run the command

   /base_relocation/**opt/ibmcmp/vacpp/6.0/bin/xlc++ -c -qinfo hello.C**

   This command emits all applicable informational messages and produces the file `hello.o`.

   Note: C++ The compiler option **-qinfo** is equivalent to **-qinfo=all:noppt**.
3. Link step. Run the command

   /base_relocation/**opt/ibmcmp/vacpp/6.0/bin/xlc++ -o hello hello.o**
4. Run the program:

   ./hello

   The expected result is ″Hello World!″ output to the screen.
5. Check the exit code of the program:

   echo $?

   The result should be zero.

Hello World with vector programming

If you have or plan to have vector programming in your source code, test a Hello World program that requires AltiVec enablement. Both the C and C++ compilers support the AltiVec C programming interface.

1. Create a C program similar to the following listing, which follows the Macintosh C Programming Model. Name the source file `hello2.c`.

   ```
   #include <stdio.h>
   int main(void)
   {
      vector unsigned char vec1 = (vector unsigned char)
            ('H','e','l','l','o',' ','W','o','r','l','d',' ','o','f',' ','C');
      printf("%vc\n", vec1);
      return 0;
   }
   ```
2. Compile.

   To invoke the C compiler, run the command

   /base_relocation/**opt/ibmcmp/vac/6.0/bin/xlc hello2.c -c -o hello2.o -qaltivec -qinfo=por**.

   To invoke the C++ compiler, run the command

   /base_relocation/**opt/ibmcmp/vacpp/6.0/bin/xlc++ hello2.c -o hello2.o -+ -qaltivec -qinfo=por**.

   Notes on the compiler options.
   - Compiling with the option **-qaltivec** automatically generates VRSAVE instructions and keeps the VRSAVE register up-to-date.

- Compiling with the option **-qinfo=por** returns the informational messages related to the language extensions for the AltiVec programming interface and to other constructs that can affect portability.
- Compiling with the option **-+** instructs the compiler to treat the input file as a C++ source file.

3. Link.

   To link using the C compiler, run the command

   /*base_relocation*/**opt/ibmcmp/vac/6.0/bin/xlc -o hello2 hello2.o**.

   To use the C++ compiler, run the command

   /*base_relocation*/**opt/ibmcmp/vacpp/6.0/bin/xlc++ -o hello2 hello2.o**

4. Run.

   ./hello2

   The expected result is to display `"Hello World of C"` to the screen.

Using the compiler for the first time

This section is a collection of notes on various topics related to using XL C/C++.

Invocation commands

Each compiler invocation command implies setting some options. The invocation command **xlc** is equivalent to explicitly setting the following options:

```
-qlanglvl=extc89 -qansialias -qcpluscmt -qkeyword=inline -qnotrigraph
```

This compiler invocation automatically defines the following IBM-specific predefined macro `__IBMC__`.

C++ The invocation command **xlc++** is equivalent to explicitly setting the following options:

```
-qlanglvl=extended -qansialias -qnotrigraph
```

This compiler invocation automatically defines the following IBM-specific predefined macro `__IBMCPP__`.

Ordinarily, the compiler uses the file name suffix as specified on the command line to determine the type of the source file: `.c` (lowercase *c*) specifies a C source file and `.C` (uppercase *c*) specifies a C++ file. For example,

`hello.c`	The file is treated as a C file.
`hello.C`	The file is treated as a C++ file.

This applies for both case-sensitive and case-insensitive file systems. However, in a case-insensitive file system, the compilations of `hello.c` and `hello.C` refer to the same physical file. That is, the compiler recognizes the case difference of the file name argument on the command line and determines the source type accordingly, but will ignore the case when retrieving the file from the file system. The **-qsourcetype** option instructs the compiler to use the source type as specified on the option, and not to rely on the case of the file name suffix.

Case sensitivity

The compiler assumes case sensitivity in its searches for files. This does not cause problems if the compiler and all its prerequiste components (especially the GNU C header files) are installed on a Mac OS X machine. The Mac OS X file system is case-insensitive, but the Mac OS X operating system also supports case-sensitive file systems.

However, if the compiler is installed on a case-sensitive file system (for example, a network configuration, in which a file system from another platform is mounted on the Mac), the mixed file system environment can cause problems for the compiler if the correct case is not specified in #include directives.

The reason is that some IBM header files replace or wrap GNU C or Mac OS X system headers and that the compiler relies on the Mac search mechanism to find a file. The replacements or wrappings work correctly in a homogeneous file system environment because case in a specified header file name is ignored. However, in a mixed environment, the IBM headers might be skipped as the operating system satisfies the search with a file of the same name but different case.

The following example assumes that the compiler has been installed in a network environment on a case-sensitive file system. If your source code specifies `#include <targetconditionals.h>`, the IBM wrapper header `TargetConditionals.h` will not be found, only the Mac OS X version of `TargetConditionals.h` will be found, and XL C/C++ compiler will not behave correctly. The reason is that the IBM file `TargetConditionals.h` is a wrapper header that uses a `#include_next` directive to include the system header file of the same name.

Therefore, a good habit is to create the files with the spelling (including capitalization) with which you intend to reference them, and that, in the source code, you consistently refer to them with the exact same spelling.

Related References
- `-qsourcetype` in *XL C/C++ for Mac OS X Compiler Reference*
- "Compiler Command Line Options" in *XL C/C++ for Mac OS X Compiler Reference*
- "Predefined Macros Related to Language Features" in *XL C/C++ for Mac OS X C/C++ Language Reference*

Customizing the compilation environment

This section discusses the mechanisms used by XL C/C++ to create a compilation environment. The mechanisms for customizing the compilation environment are environment variables, the names and locations of include files, attributes in a configuration file, and command-line options. For example, if you set search paths for include files and libraries and specify the location of gcc, you create a particular compilation environment. The vacpp_configure utility, which facilitates the creation of valid configuration files, is also discussed in this section.

The important search paths for XL C/C++ are the standard directory locations for:
- The GNU C compiler
- GNU C include files
- GNU C++ include files
- IBM C headers
- GNU C library paths

Environment variables

Part of the compilation environment are the search paths for special files such as libraries and include files. The following system variables are used by the compiler.

DYLD_LIBRARY_PATH	Specifies the directory path for dynamically loaded libraries. Used by the system linker at link time and at run time.
MANPATH	Specifies the search path for system and third-party software man pages.
NLSPATH	Specifies the directory path of National Language Support libraries.
PATH	Specifies the directory path for the executable files of the compiler.
PDFDIR	Specifies the directory in which the profile data file is created. The default value is unset, and the compiler places the profile data file in the current working directory. Setting this variable to an absolute path is recommended for profile-directed feedback.
TMPDIR	Specifies the directory in which temporary files are created. The default location may be inadequate at high levels of optimization, where temporary files can require significant amounts of disk space.

Create symbolic links for the PATH

The command-line interfaces for XL C/C++ for Mac OS X are not automatically installed in /usr/bin. To invoke the compiler without having to specify the full path, do one of the following steps:
- Create symbolic links for the specific driver contained in /opt/ibmcmp/vacpp/6.0/bin and /opt/ibmcmp/vac/6.0/bin to /usr/bin.
- Add /opt/ibmcmp/vacpp/6.0/bin and /opt/ibmcmp/vac/6.0/bin to the PATH environment variable.

Ensuring the correct NLSPATH

If you used relocations when you installed the run-time libraries for the compiler or SMP, you must set the NLSPATH to reflect the new locations of the message catalogs. During compilation, the compiler configuration file provides paths to the correct message catalogs. However, at run-time, a search for message catalogs uses the NLSPATH environment variable.

The following table shows the paths you must add for each relocatable package.

Updating the NLSPATH

Package	Addition to NLSPATH
xlsmp.msg.rte.pkg, xlsmp.rte.pkg, xlsmp.lib.pkg	`*smprt-path*/opt/ibmcmp/msg/en_US/%N`
vac.lic.pkg	(no change)
vac.cmp.pkg	(no change)
vacpp.cmp.pkg	`*relocation-path*/opt/ibmcmp/vacpp/6.0/msg/en_US/%N`
vacpp.rte.pkg	(no change)
vac.help.pkg	(no change)
vacpp.help.pkg	(no change)
vac.samples.pkg	(no change)

Include files

The locations for GNU, IBM, and system header files are most conveniently specified in a configuration file.

The compiler option **-I** *directory_name* allows you to add directories to the search paths in the configuration file. The configuration file itself uses the -I option internally to set the directory paths that it controls. The compiler searches the directories specified by -I within the configuration file before searching those specified by -I options on the command line.

See *XL C/C++ for Mac OS X Compiler Reference* for more information.

Configuration files

A configuration file is a plain text file that specifies options that are read every time you run the compiler. The name of a configuration file ends with a .cfg file name extension.

After you have a default configuration file (/etc/opt/ibmcmp/vac/6.0/vac.cfg), you can create others. XL C/C++ for Mac OS X provides a template, which can be used to create /etc/opt/ibmcmp/vac/6.0/vac.cfg. However, any existing configuration file can be used as the template for creating another with the vacpp_configure utility.

You can instruct the compiler to use a particular configuration file by invoking the compiler with the **-F** option and specifying the fully qualified file name.

The vacpp_configure utility

To achieve binary compatibility with gcc-compiled code, the XL C/C++ compiler requires at least one GNU compiler to be present on the system in order to use its header files, libraries, and some of its binary utilities. A configuration file saves the location of the GNU compiler that the XL C/C++ compiler should use. The specification is necessary because the installation of a GNU compiler does not mandate a location and because more than one instance of the GNU compiler can exist on the same system. Consequently, XL C/C++ needs to know which to use and to be able to find it and its related headers, libraries, and utilities on the system.

The directory paths to these important resources are saved as the values of attributes in the configuration file. The following table describes these crucial attributes.

Platform-specific configuration attributes

Attribute name	**Usage**
gcc_path	Specifies the location of the **gcc** command. The link command uses the value of this attribute to link C applications.
gcc_c_stdinc	Specifies the standard locations used by gcc to search for the include files for C programs. The value is a colon-separated list of directory paths. The user can override the list by compiling with the **-qgcc_c_stdinc=** option.
gcc_cpp_stdinc	Specifies the standard locations used by gcc to search for the include files for C++ programs. The value is a colon-separated list of directory paths. The user can override the list by compiling with the **-qgcc_cpp_stdinc=** option.
gcc_libs	A comma-separated list of the GNU C libraries.
gcc_libdirs	A comma-separated list of directories that contain GNU C libraries.
xlc_c_stdinc	Specifies the list of product-specific search locations for C programs. The value is a colon-separated list of directory paths. The user can override the list by compiling with the **-qc_stdinc=** option.

The **vacpp_configure** utility facilitates creating and updating configuration files.

The syntax for **vacpp_configure** is as follows:

```
                                                      ┌─────────────────────┐
                                                      ▼                     │
►►──vacpp_configure──┬──────────────┬──┬──────────┬──-gcc──────gcc_path─────┴──────────────►
                     ├──-o──name────┤  └──-force──┘
                     └──-install────┘

►──┬────────────────────────────────────────────────────────────────────┬──────────────►
   ├──-ibmcmp──path_base────────────────────────────────────────────────┤
   └──┤ -smprt ├──┤ -vac ├──┤ -vacpprt ├──┤ -vacpp ├──┤ -vaclic ├───────┘

►──┬────────────────────────────┬─────────────────────────────────────────────────────►◄
   └──template_config_filename──┘
```

-smprt:

```
├──┬─────────────────────────────────────────┬──────────────────────────────────────┤
   └──-smprt──smprt_path/opt/ibmcmp/xlsmp/1.4──┘
```

-vac:

```
├──┬─────────────────────────────────────────┬──────────────────────────┤
   └─-vac──relocation_path/opt/ibmcmp/vac/6.0─┘
```

-vacpprt:

```
├──┬────────────────────────────────────────────────────────────┬───────────┤
   └─-vacpprt──vacpprt_relocation_path/opt/ibmcmp/vacpp/6.0─┘
```

-vacpp:

```
├──┬─────────────────────────────────────────────┬──────────────────────┤
   └─-vacpp──relocation_path/opt/ibmcmp/vacpp/6.0─┘
```

-vaclic:

```
├──┬───────────────────────────────────────────────┬────────────────────┤
   └─-vaclic──license_relocation/opt/ibmcmp/vac/6.0─┘
```

where

-o *name*
: Specifies the name of the configuration file to generate. By default, output is written to `stdout`.

-force
: Forces the vacpp_configure utility to overwrite any existing output file with the specified name and path. By default, vacpp_configure issues an error and stops if the specified file already exists.

-install
: Equivalent to specifying **-o /etc/opt/ibmcmp/vac/6.0/vac.cfg**.

-gcc *gcc_path*
: Specifies the parent of the bin directory where gcc is installed. The gcc command is assumed to be in the bin directory under this path. In the configuration file, the attribute `gcc_path` is set equal to this path. This attribute can be specified a maximum of one time.

  For example, if the gcc command is located in /usr/bin/gcc, you would specify

  ```
  -gcc /usr
  ```

-smprt *smprt_path***/opt/ibmcmp/xlsmp/1.4**
: Specifies the path to the SMP run-time libraries. If you installed to the default location, the path is /opt/ibmcmp/xlsmp/1.4. If you installed to a relocation, the path must end with opt/ibmcmp/xlsmp/1.4.

-vac *relocation_path***/opt/ibmcmp/vac/6.0**
: Specifies the path to the C compiler. If you installed to the the default installation location, the path is /opt/ibmcmp/vac/6.0. If you installed to a relocation, the path must end with opt/ibmcmp/vac/6.0.

-vacpprt *vacpprt_relocation_path***/opt/ibmcmp/vacpp/6.0**
: Specifies the path to the C++ run-time libraries. If you installed to the default relocation, you should use the value /opt/ibmcmp/vacpp/6.0. If you installed to a relocation, the path must end with opt/ibmcmp/vacpp/6.0.

-vacpp *relocation_path***/opt/ibmcmp/vacpp/6.0**
: Specifies the path to the C++ compiler.

-vaclic *license_relocation***/opt/ibmcmp/vac/6.0**
: Specifies the path to `vac.lic.pkg`, the licensing package.

-ibmcmp *path_base*
: Specifies the path to the XL C/C++ compilers. The specified path *path_base* replaces /opt/ibmcmp in the template configuration file. Use this option to indicate the relocation for all packages. If you installed to the default relocation, the value of *path_base* is /opt/ibmcmp. If you installed to a relocation, the value of *path_base* must end with opt/ibmcmp.

template_config_filename
: The input file that is used to construct a configuration file. The default template is /opt/ibmcmp/vac/6.0/etc/vac.base.cfg.

For example, the command

```
vacpp_configure -o /home/local/myconfigfile.cfg -gcc /usr/local/gcc3
```

uses the template /opt/ibmcmp/vac/6.0/etc/vac.base.cfg to create the configuration file /home/local/myconfigfile.cfg. The gcc_path attribute in the configuration file has stored /usr/local/gcc3 as the parent of the bin directory that contains the gcc compiler, which means that gcc is installed under /usr/local/gcc3/bin.

Command-line options

The compiler options controlling the standard include paths are shown in the table below. The value for *paths* is specified by the user on the command line. When a configuration file is used, the value for *paths* is the value of the attribute named in the second column. The configuration file created by the vacpp_configure utility is used by default. The compiler processes each attribute in the configuration file and creates a corresponding option to set the proper search path for include files.

Platform-specific configuration options and related attributes

Option name	Attribute	Usage	Conflict resolution
-qgcc_c_stdinc=<*paths*>	gcc_c_stdinc	Specifies the search locations for the gcc headers. The default value is the value of the attribute in the configuration file.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qgcc_cpp_stdinc=<*paths*>	gcc_cpp_stdinc	Specifies the search locations for the g++ headers. The default value is the value of the attribute in the configuration file.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qc_stdinc=<*paths*>	xlc_c_stdinc	Specifies the search locations for standard include files for the IBM C headers. The default value is the value of the attribute in the configuration file.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qcpp_stdinc=<*paths*>	xlc_cpp_stdinc	Specifies the search locations for the IBM C++ headers. The default value is the value of the attribute in the configuration file.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.

Related References
- "Platform-specific options" on page 30

Using XL C/C++ with Xcode and Project Builder

IBM XL C/C++ has been developed to be compatible with the GNU C and C++ compilers, Version 3.3, on the Mac OS X platform. In addition to having a command-line interface, XL C/C++ can be used in the Apple integrated development environments (IDEs) Xcode and Project Builder.

New projects created in the Xcode IDE use the native build system. On the other hand, Project Builder uses the `jam` utility, which is similar to the `make` utility commonly used on other UNIX platforms. The XL compilers do not support a `jam`-based build from within Xcode. You can either upgrade an existing Project Builder project to a native Xcode project, or you can use the XL compilers with Project Builder to build the project.

Configure the Xcode IDE

The Xcode IDE uses specification files to describe the compilers and associated utilities that are used in a build. During the XL compiler installation, the compiler specification file XLC.6.0.pbcompspec is copied into the /Library/Application Support/Apple/Developer Tools/Specifications directory. This file does not change your existing Xcode settings. You can continue to use `gcc` as your build compiler, or you can use XL C/C++, as described in the next section.

Using Xcode with XL C/C++

This section describes how to build projects using XL C/C++.

The following steps describe how to modify the Xcode IDE settings to use the utilities **gxlc** and **gxlc++** to compile C or C++ source files. These utilities translate GNU compiler options into XL C/C++ equivalents and invoke the XL compilers. Many of the commonly used GNU C and C++ compiler options have been mapped to equivalent XL C/C++ options. However, not all GNU C and C++ options have an XL C/C++ equivalent. See *Related References*.

Any XL C/C++ compiler option that has no corresponding **gxlc** or **gxlc++** equivalent must be prefixed with '`-Wx,`' when specified for **gxlc** or **gxlc++**. For example, to use the `-qnolibansi` compiler option, specify `-Wx,-qnolibansi`.

The following steps describe how to set up an Xcode project that uses the XL C and C++ compilers:

1. Start the Xcode IDE.
2. Create a new project or open an existing project. Ensure that you have write permission on the project directory.
3. Select the project in the **Groups & Files** list.
4. Click the **Info** button on the toolbar to display the **Info** window of the project.
5. Click the **Styles** tab to open the **Styles** pane of the project.
6. Clear the **Zero Link** and **Fix & Continue** options if they are selected. XL C/C++ does not support them.
7. Close the **Info** window of the project.
8. Select the target that is being configured from the **Groups & Files** list.
9. Click the **Info** button on the toolbar to display the **Info** window of the target.
10. Click the **Rule** tab.
11. Click the plus (+) button at the bottom left of the **Rule** pane to add a new rule.

12. From the **Process** list, specify the type of source files for the new rule. To use the XL compilers select **C source files**. This rule applies to both C and C++ source files.
13. From the **Using** list, select **IBM XL C/C++ Version 6.0**. This list item informs the IDE to use the IBM C or C++ compiler, depending on the type of the source files. Alternatively, you can select **IBM XLC C Version 6.0**; if your project contains only C source files. Both list items will work for a project that uses C source files.
14. Click the **Build** tab.
15. Clear the **Precompile prefix header** if it is selected. XL C/C++ does not support it.
16. If you are building a C++ application, clear the **Prebinding** checkbox under Standard Build Settings. Currently the XL C++ run time does not support prebinding.
17. Add compiler options:
    - Specify GNU C and C++ compiler options in the Build Pane of the target. Only the GNU options that can be translated by **gxlc** or **gxlc++** can be specified this way. For those that cannot be translated, specify them as XL C/C++ compiler options as described in the next bullet.
    - Specify XL C/C++ compiler options by adding them to **Current Settings/OTHER_CFLAGS**, using the format `-Wx,-`*`ibm_option`*
    - Specify any XL C/C++ linking options in **Standard Build Settings/Linking/Other linker flags**, using the format `-Wx,-`*`ibm_option`*
18. Close the **Info** window of the target.
19. Build the project.

Hints and tips for using XL C/C++ with the Xcode IDE

- By default, the XL C/C++ compilers search for headers files in the same directory as the current source file. If there are separate header file directories, you need to specify them in ″Header search paths″ under the Build pane of the target.
- If your application includes `Carbon.h` using `#include <Carbon.h>` instead of `#include <Carbon/Carbon.h>`, you need to add the header file search path /Developer/Headers/FlatCarbon to the Header search paths under the Build pane of the target.
- Most of the warning options provided under the GNU option settings are not supported by the XL C/C++ compilers. These are the GNU warning -W options. You can get similar compiler diagnostics by using XL C/C++ suboptions of -qinfo.
- The macro `__APPLE_CC__` is not defined by the XL C/C++ compilers.

Related References
- "GNU C and C++ to XL C/C++ option mapping" on page 33

Using Project Builder with XL C/C++

You can redirect the Project Builder IDE to use **gxlc** and **gxlc++** to translate a GNU compiler option into an XL C/C++ equivalent. The scripts **xlc_pb** and **xlc++_pb** provide the necessary interface to invoke **gxlc** and **gxlc++** respectively.

Any XL C/C++ compiler option that has no corresponding **gxlc** or **gxlc++** must be prefixed with `-Wx,`. For example, to use the `-qnolibansi` compiler option, specify `-Wx,-qnolibansi`.

The following steps describe how to use Project Builder with the XL compilers. These steps can only be used on C or C++ projects.

1. Start Project Builder.
2. Create a new project or open an existing project.
3. Edit the build settings for the target as follows:
   - Set the CC and CPLUSPLUS variables. See next section for step-by-step instructions.
   - Specify GNU C or C++ compiler options.
   - Specify XL C/C++ compiler options in the format `-Wx,-`*`ibm_option`*
4. Build the project.

Related References
- "GNU C and C++ to XL C/C++ option mapping" on page 33

Setting the CC and CPLUSPLUS variables

The scripts **xlc_pb** and **xlc++_pb** are used to provide the interface to Project Builder. They do the necessary setup and then run **gxlc** or **gxlc++**. To direct Project Builder to use these scripts, you must set the CC variable if the project contains C source files and the CPLUSPLUS variable if the project contains any C++ source files. You need to do this only once for a project.

To set the CC or CPLUSPLUS variables, do the following:
1. Open an existing project or create a new one.
2. Click the **Targets** tab to display a contents list.
3. In the contents list, under Targets, select the name of the target you are working on. The Target Summary pane appears.
4. From the navigation tab on the left, select **Settings > Expert View**. A list of Build Settings appears.
5. Add an entry to the Build Settings by clicking the plus sign (+) at the bottom left of the list.
6. In the Name field, rename the new setting **CC**. In the Value field, enter the full path to the script **xlc_pb**. The script is located in the *$vacPath*/exe/ directory. For example, if you used the default installation location, the value you should enter is: /opt/ibmcmp/vac/6.0/exe/xlc_pb.
7. Click the plus sign (+) at the bottom left of the Build Settings list to add an entry for the CPLUSPLUS variable.
8. In the Name field, rename the setting **CPLUSPLUS**. In the Value field, enter the full path to the script **xlc++_pb**. The script is located in the *$vacppPath*/exe/ directory. For example, if you used the default installation location, the value you should enter is: /opt/ibmcmp/vacpp/6.0/exe/xlc++_pb.

Hints and tips for using XL C/C++ with Project Builder

See "Hints and tips for using XL C/C++ with the Xcode IDE" on page 23.

Controlling the compilation process

The overall compilation process consists of three phases: preprocessing, translation to object code, and linking. By default, a compiler invocation command invokes all phases of the compilation process to translate a program from source code to executable output. If file names for input and output files are specified when the compiler is invoked, it determines the starting and ending phases from the file name suffix (extension) of the input and output files.

You can also create a particular type of output file at any compilation phase by using appropriate compiler options. For example, invoking the **xlc** or **xlc++** command with the `-E` or `-P` option performs only the preprocessing phase on the input files. The compiler invocation determines from the extension of the input file name whether to call the IBM compiler, the system linker, or the Mach-O assembler.

Invoking the compiler

In most cases, you should use the **xlc++** command to compile C++ source files and the **xlc** command to compile C source files.. Variations of the basic compiler invocation command exist primarily to support different version levels of the C language and different language extensions for C++. Both **xlc** anc **xlc++** will compile source as either C or C++, but compiling C++ files with **xlc** may result in link errors or run-time errors. The errors result because all the libraries required for C++ code are not specified when the linker is called.

Compiler invocation commands

Basic invocation command	**Thread-safe variation**	**Description**
xlc++	**xlc++_r**	Invokes the compiler so that source files are compiled as C++ language source code. Equivalent to **xlC** and **xlC_r**.
xlC	**xlC_r**	Invokes the compiler so that source files are compiled as C++ language source code.
xlc	**xlc_r**	Invokes the compiler to compile C source files with the default language level of C89 and the compiler options **-qlonglong** and **-qansialias**.
c89	**c89_r**	Invokes the compiler to compile enforcing strict conformance to the ISO C89 standard (ISO/IEC 9899:1990).
c99	**c99_r**	Invokes the compiler with support for C99 features. Note that full conformance with the ISO C99 standard (ISO/IEC 14882:1998) also requires support from the run-time library.
cc	**cc_r**	Invokes the compiler for use with legacy C code that does not require compliance with C89 or C99.
gxlc		Invokes the compiler after translating gcc command-line options to XL C/C++ options. Note that not every gcc option has an exact XL C/C++ equivalent.
gxlc++		Invokes the compiler after translating g++ command-line options to XL C/C++ options. Note that not every g++ option has an exact XL C/C++ equivalent.

Types of input and output files

The compiler uses the file name extension to determine the appropriate compilation phase and invoke the associated tool.

The compiler accepts the following types of files as input:

Accepted input file types

File type description	File name extension	Example
C and C++ source file	.c (lowercase *c*) for C language source files; .C (uppercase *c*), .cc, .cp, .c++, .cpp, .cxx for C++ source files	*file_name*.c *file_name*.C, *file_name*.cc, *file_name*.cpp, *file_name*.cxx
Preprocessed source file	.i	*file_name*.i
Object file	.o	`hello.o`
Assembler file	.s	`check.s`
Archive file	.a	`v1r5.a`
Shared library file	.dylib	`libFoo.dylib`
IPA control files (**-qipa=***file_name*)	No naming convention for *file_name* is enforced.	`ipa.ctl`

Note: To control the interpretation of input files with the file name extension .C (capital *c*), use the compiler option `-qsourcetype`. This option provides the ability to override the source file type implied by the file name extension.

You can specify the following types of output files when invoking the compiler:

Types of output files

File type description	Example
Executable file	By default, `a.out`
Object files	*file_name*.o
Preprocessed files	*file_name*.i
Listing files	*file_name*.lst
Target file	*file_name*.d

Related References
- `-qsourcetype` in *XL C/C++ for Mac OS X Compiler Reference*

Default behavior

If you invoke the compiler without specifying any options, the behavior of the compiler is governed by the following default settings:
- Attempts to read and invoke the options specified in a configuration file.
- Searches for library files.
- Aligns structures using `-qalign=power` alignment: uses 8 bytes as the strictest alignment constraint for structures and 16-byte alignment for AltiVec vector types.

- Produces an unoptimized executable named `a.out`.
- Diagnoses AltiVec programming constructs as syntax errors.

See *XL C/C++ for Mac OS X Compiler Reference* for more information.

Getting started with compiler options

Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions. You can specify compiler options on the command line, in a configuration file, in your source code, or any combination of these techniques. Most options that are not explicitly set take the default settings.

When multiple compiler options have been specified, it is possible for option conflicts and incompatibilities to occur. To resolve these conflicts in a consistent fashion, the compiler applies the following priority sequence unless otherwise specified:

1. Source file *overrides*
2. Command line *overrides*
3. Configuration file *overrides*
4. Default settings

Generally, among multiple command-line options, the last specified prevails.

Note: The **-I** compiler option is a special case. The compiler searches any directories specified with **-I** in the vac.cfg file *before* it searches the directories specified with **-I** on the command line. The option is cumulative rather than preemptive.

The option **-l** (lowercase L) also has cumulative behavior. See *XL C/C++ for Mac OS X Compiler Reference* for more information.

Compiler messages

XL C/C++ uses a five-level classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. Not every error halts compilation. The following table provides a key to the abbreviations for the severity levels and the associated compiler response.

Severity levels and compiler response

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.
W	Warning	Compilation continues. The message reports valid but possibly unintended conditions.
▸ C E	Error	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not produce the expected results.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.

Severity levels and compiler response

Letter	Severity	Compiler Response
U	Unrecoverable error	The compiler halts. An unrecoverable error has been encountered. If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. If it indicates that different compiler options are needed, recompile using them. If the message indicates an internal compiler error, the message should be reported to your IBM service representative.

The default behavior of the compiler is to compile with the option **-qnoinfo** or **-qinfo=noall**. The suboptions for **-qinfo** provide the ability to specify a particular category of informational diagnostics. For example, **-qinfo=por** limits the output to those messages related to portability issues.

Note: In C, the option **-qinfo** is equivalent to **-qinfo=all**; in C++, **-qinfo** is equivalent to **-qinfo=all:noppt**.

Return codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:
- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the **-qhalt** compiler option, and the number of errors did not reach the limit set by the **-qmaxerr** compiler option.
- No message specified by the **-qhaltonmsg** compiler option is issued.

Otherwise, the compiler sets one of the return codes documented in *XL C/C++ for Mac OS X Compiler Reference*.

Compiler message format

By default, diagnostic messages have the following format:

```
"file", line line_number.column_number: 15cc-nnn (severity) message_text.
```

where 15 is the compiler product identifier, `cc` is a two-digit code indicating the compiler component that issued the message (for example, compiler, linker, assembler), `nnn` is the message number, and `severity` is the letter of the severity level.

This format is the same as compiling with the **-qnosrcmsg** option enabled. To get an alternate message format in which the source line displays with the diagnostic message, try compiling with **-qsrcmsg** option.

Note: Messages are not intended to be used as input to other programs. The message format and content are not intended to be a programming interface and may change from release to release.

Platform-specific options

This section features options that are basic to using the compiler on the Mac OS X platform.

See *XL C/C++ for Mac OS X Compiler Reference* for more information.

Selected compiler options specific to the Mac OS X platform

Option name	Description
`-qgcc_c_stdinc=<`*`paths`*`>`	Specifies the directory search paths for the GNU C headers.
`-qgcc_cpp_stdinc=<`*`paths`*`>`	Specifies the directory search paths for the GNU C++ headers.
`-qc_stdinc=<`*`paths`*`>`	Specifies the directory search paths for the IBM C headers.
`-qcpp_stdinc=<`*`paths`*`>`	Specifies the directory search paths for the IBM C++ headers.
`-qalign`	Specifies the alignment rules for aggregates. The default is `-qalign=power`
`-qaltivec`	Enables support for AltiVec vector programming.
`-qarch`	Specifies the architecture on which the executable program will run. The default is `-qarch=ppcv`.
`-qchars`	Instructs the compiler to treat all variables **char** as either signed or unsigned. The default is `-qchars=signed`.
`-qcommon`	Indicates the preferred section into which the compiler should place uninitialized global data. The default is `-qcommon`, meaning the .comm section. `-qnocommon` requests the compiler to use the .data section.
`-qcomplexgccincl`	Instructs the compiler to place `#pragma complexgcc(on)` and `#pragma complexgcc(pop)` directives around include files in the specified directories. The default is provided by the configuration file specified at compile time.
`-qframeworkdir`	Specifies the path to framework directories.
`-qmacpstr`	Enables Pascal-style string handling. The default is `-qnomacpstr`.
`-qpic`	Instructs the compiler to produce position-independent code. The default is `-qpic`.
`-qsourcetype`	Controls the interpretation of input file names. The default behavior is that the programming language of a source file is implied by the suffix of its file name.
`-qstdframework`	Instructs the compiler to search the standard framework directories. The default is `-qstdframework`.
`-qtrigraph`	Instructs the compiler to interpret or not interpret trigraph sequences, regardless of the specified language level. The default on the Mac OS X platform is `-qnotrigraph`, which enables the expected interpretation of the character literal '????' often used for version checking.
`-qtune`	Specifies the architecture for which the executable program is optimized. The default is `-qtune=ppc970`.
`-qvrsave`	Specifies that prologs and epilogs of functions in the compilation unit should include code needed to maintain the VRSAVE register. The default is `-qvrsave`.
`-qwarnfourcharconsts`	Instructs the compiler to generate warning messages for four-character constants. The default is `-qnowarnfourcharconsts`.

The following options provide specialized control of directory search paths. The options are cumulative, rather than preemptive. The paths specified on the command line with the options **-L** and **-l** (lowercase *L*) will have lower priority at link time than those specified as an option in the configuration file, but higher priority than paths specified as an attribute in the configuration file.

Selected path control options

Option name	Description
-I	Specifies additional directory paths to be searched for `#include` files.
-L	Specifies the library search paths to be searched at link time.

Related References

- "Options summary: C++ compiler" on page 40
- "Compiler Command Line Options" in *XL C/C++ for Mac OS X Compiler Reference*.

Reusing GNU C and C++ compiler options with gxlc and gxlc++

Each of the **gxlc** and **gxlc++** utilities accepts GNU C and C++ compiler options and translates them into comparable XL C/C++ options. Each utility uses the XL C/C++ options to create an **xlc** or **xlc++** invocation command, which it then uses to invoke XL C/C++. These utilities are provided to facilitate the reuse of make files created for applications previously developed with GNU C and C++. However, to fully exploit the capabilities of XL C/C++, it is recommended that you use the XL C/C++ invocation commands and their associated options.

The actions of **gxlc** and **gxlc++** are controlled by the configuration file `gxlc.cfg`. Both utilities use the same configuration file. The GNU C and C++ options that have an XL C or XL C++ counterpart are shown in this file. Not every GNU C and C++ option has a corresponding XL C/C++ option. The **gxlc** utility returns a warning for any GNU C option it cannot translate; similarly, **gxlc++** for any g++ options that are not mapped to a corresponding XL C++ option. The **gxlc** and **gxlc++** option mappings are modifiable. For information on adding to or editing the **gxlc** and **gxlc++** configuration file, see "Configuring the option mapping" on page 36.

Example

To use the gcc `-ansi` option to compile the C version of the Hello World program, you can use:

```
gxlc -ansi hello.c
```

which translates into:

```
xlc -qlang=extc89 hello.c
```

This command is then used to invoke the XL C compiler.

gxlc and gxlc++ return codes

Because **gxlc** and **gxlc++** invoke the compiler, they return output, like any other invocation method. In addition to listings, diagnostic messages related to the compilation, and return codes, **gxlc** and **gxlc++** return warnings for input options that were not translated. If gxlc or gxlc++ cannot successfully call the compiler, it sets the return code to one of the following values:

40	A gcc or g++ option error or unrecoverable error has been detected.
255	An error has been detected while the process was running.

gxlc and gxlc++ syntax

The following diagram shows the **gxlc** and **gxlc++** syntax:

```
►►──┬─gxlc───┬──┬──────┬──┬─────────────────────────────┬──┬──────────────────────┬──filename───────────►◄
    └─gxlc++─┘  ├─-v───┤  └─-Wx,──xlc_or_xlc++_options─┘  └─gcc_or_g++_options─┘
                └─-vv──┘
```

where:

filename
: Is the name of the file to be compiled.

-v Allows you to verify the command that will be used to invoke XL C/C++. gxlc or gxlc++ displays the XL C/C++ invocation command that it has created, before using it to invoke the compiler.

-vv Allows you to run a simulation. gxlc or gxlc++ displays the XL C/C++ invocation command that it has created, but does not invoke the compiler.

-Wx,*xlc_or_xlc++_options*
Sends the given XL C/C++ options directly to the `xlc` or `xlc++` invocation command. gxlc or gxlc++ adds the given options to the XL C/C++ invocation it is creating, without attempting to translate them. Use this option with known XL C/C++ options to improve the performance of the utility.

gcc_or_g++_options
Are the gcc or g++ options that are to be translated to `xlc` or `xlc++` options. The utility emits a warning for any option it cannot translate. The gcc and g++ options that are currently recognized by gxlc and gxlc++ are listed in the configuration file `gxlc.cfg`.

GNU C and C++ to XL C/C++ option mapping

The following table lists the gcc options that are accepted and translated by **gxlc** and **gxlc++**. All other GNU C and C++ options that are specified as input to **gxlc** and **gxlc++** are ignored or generate an error. If the negative form of a GNU C and C++ option exists, then the negative form is also recognized and translated by **gxlc** and **gxlc++**.

Table 1. Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
-###	-#
C -ansi	-qlang=extc89
-C	
-c	
-Dmacro*[=defn]*	
-E	
-e	
-F*[dir]*	-qframeworkdir=*dir*
-faltivec	-qaltivec
-fcommon	-qcommon
-fdollars-in-identifiers	-qdollar
C++ -fdump-class-hierarchy	-qdump_class_hierarchy
C++ -fexceptions	-qeh
C++ -ffor-scope	-qlanglvl=ansifor
C++ -fno-for-scope	-qlanglvl=noansifor
-finline	-qinline
-finline-functions	-qinline
-finline-limit=*n*	-qinline=*n*
C++ -fkeep-inline-functions	-qkeepinlines

Table 1. Mapped options: GNU C and C++ to XL C/C++ (continued)

GNU C and C++ option	XL C/C++ option
-fno-asm	-qnokeyword=asm -qnokeyword=inline -qnokeyword=typeof
C++ -fno-gnu-keywords	-qkeyword=typeof
C++ -fno-operator-names	-qkeyword=and -qnokeyword=bitand -qnokeyword=bitor -qnokeyword=compl -qnokeyword=not -qnokeyword=or -qnokeyword=xor
-fpascal-strings	-qmacpstr
-fPIC	-qpic
C++ -frtti	-qrtti
-fshort-enums	-qenum=small
-fsigned-bitfields	-qbitfields=signed
-fsigned-char	-qchars=signed
-fstrict-aliasing	-qansialias
-fsyntax-only	-qsyntaxonly
-funroll-all-loops	-qunroll=all
-funroll-loops	-qunroll=all
-funsigned-bitfields	-qbitfields=unsigned
-funsigned-char	-qchars=unsigned
-fwritable-strings	-qnoro
-g	
-g3	-g
-ggdb	-g
-gstabs	-g
-I*dir*	
-L*dir*	
-l*library*	
-M	
-MD	-M
-malign-mac68k	-qalign=mac68k
-malign-natural	-qalign=natural
-malign-power	-qalign=power
-dynamic-no-pic	-qnopic
-mno-fused-madd	-qfloat=nomaf
-mfused-madd	-qfloat=maf
-nodefaultlibs	-qnolib
-nostartfiles	-qnocrt
-nostdinc	-qnostdinc
-nostdlib	-qnolib -qnocrt
-O	

Table 1. Mapped options: GNU C and C++ to XL C/C++ (continued)

GNU C and C++ option	XL C/C++ option
-O0	-qnoopt
-O1	-O
-O2	
-O3	
-Os	-O -qcompact
-o	
-p	
-pg	
-r	
-s	
▶ C -std=c89	-F:c89
▶ C -std=iso9899:1990	-F:c89
▶ C -std=iso9899:199409	-F:c89
▶ C -std=c99	-F:c99
▶ C -std=c9x	-F:c99
▶ C -std=iso9899:1999	-F:c99
▶ C -std=iso9899:199x	-F:c99
▶ C -std=gnu89	-qlang=extc89
▶ C -std=gnu99	-qlang=extc99
▶ C -std=gnu9x	-qlang=extc99
▶ C++ -std=c++98	-qlang=strict98
▶ C++ -std=gnu++98	-qlang=extended
-U*macro*	
-u	
-Wuninitialized	-qinfo=ini
-Wunreachable-code	-qinfo=eff
-Wa,*option*	
-Wfour-char-constants	-qwarnfourcharconsts
-Wl,*option*	
-Wp,*option*	
-w	
-x assembler	-qsourcetype=assembler
-x c	-qsourcetype=c
-x c++	-qsourcetype=c++
-x none	-qsourcetype=default
-x objective-c	-qsourcetype=objc
-Z	

Configuring the option mapping

The **gxlc** and **gxlc++** utilities use the configuration file `gxlc.cfg` to translate GNU C and C++ options to XL C/C++ options. Each entry in `gxlc.cfg` describes how gxlc or gxlc++ should map a GNU C and C++ option to an XL C/C++ option and how to process it. The following description of how to configure the option mapping pertains equally to gxlc++ as well as gxlc.

An entry consists a string of flags for the processing instructions, a string for the GNU C option, and a string for the XL C/C++ option. The three fields must be separated by whitespace. If an entry contains only the first two fields and the XL C/C++ option string is omitted, the GNU C option in the second field will be recognized by gxlc and silently ignored.

The `#` character is used to insert comments in the configuration file. A comment can be placed on its own line, or at the end of an entry.

The following syntax is used for an entry in `gxlc.cfg`:

```
abcd    "gcc_or_g++_option"    "xlc_or_xlc++_option"
```

where:

a
: Lets you disable the option by adding `no-` as a prefix. The value is either `y` for yes, or `n` for no. For example, if the flag is set to `y`, then `fcommon` can be disabled as `fno-common`, and the entry is:

```
ynn*       "-fcommon"                "-qcommon"
```

If given `-fno-common`, then gxlc will translate it to `-qnocommon`

b
: Informs gxlc that the XL C/C++ option has an associated value. The value is either `y` for yes, or `n` for no. For example, `finline-limit=`*n* maps to `qinline=`*n*. In this case, the flag is set to `y`, and the entry is:

```
nyn*       "-finline-limit"          "-qinline"
```

gxlc will then expect a value for these options.

c
: Controls the processing of the options. The value can be:

- `n`, which tells gxlc to process the option listed in the gcc-option field
- `i`, which tells gxlc to ignore the option listed in the gcc-option field. gxlc will generate an informational message indicating that this has been done, and continue processing the given options.
- `w`, which tells gxlc to ignore the option listed in the gcc-option field. gxlc will generate a warning message indicating that this has been done, and continue processing the given options.
- `e`, which tells gxlc to halt processing if the option listed in the gcc-option field is encountered. gxlc will also generate an error message.

For example, the gcc option `I-` is not supported and must be ignored by gxlc. In this case, the flag is set to `i`, and the entry is:

```
nni*       "-I-"
```

If gxlc encounters this option as input, it will not process it and will generate an informational message.

d Lets gxlc include or ignore an option based on the type of compiler. The value can be:

- `c`, which tells gxlc to translate the option only for C.
- `x`, which tells gxlc to translate the option only for C++.
- `*`, which tells gxlc to translate the option for C and C++.

For example, `fwritable-strings` is supported by both compilers, and maps to `qnoro`. The entry is:

```
nnn*       "-fwritable-strings"       "-qnoro"
```

"gcc_or_g++_option"
Is a string representing a gcc or g++ option supported by GNU C, Version 3.3. This field is required and must appear in double quotation marks.

"xlc_or_xlc++_option"
Is a string representing an XL C or XL C++ option. This field is optional, and, if present, must appear in double quotation marks. If left blank, gxlc or gxlc++ ignores the *gcc_or_g++_option* in that entry.

It is possible to create an entry that will map a range of options. This is accomplished by using the asterisk (`*`) as a wildcard. For example, the gcc `D` option requires a user-defined name and can take an optional value. It is possible to have the following series of options:

```
-DCOUNT1=100
-DCOUNT2=200
-DCOUNT3=300
-DCOUNT4=400
```

Instead of creating an entry for each version of this option, the single entry is:

```
nnn*       "-D*"                   "-D*"
```

where the asterisk will be replaced by any string following the `-D` option.

Conversely, you can use the asterisk to exclude a range of options. For example, if you want gxlc or gxlc++ to ignore all the `std` options, then the entry would be:

```
nni*      "-std*"
```

When the asterisk is used in an option definition, option flags *a* and *b* are not applicable to these entries.

The character `%` is used with a GNU C or g++ option to signify that the option has associated parameters. This is used to insure that gxlc or gxlc++ will ignore the parameters associated with an option that is ignored. For example, the `include` option is not supported and uses a parameter. Both must be ignored by the application. In this case, the entry is:

```
nni*       "-include %"
```

Related References
- "Options summary: C++ compiler" on page 40
- *XL C/C++ for Mac OS X Compiler Reference*
- The GNU Compiler Collection online documentation at `http://gcc.gnu.org/onlinedocs/`

Options summary: C compiler

This chapter appendix presents a summary of the C compiler options, grouped by type. The higher level groupings contain subgroups of options for basic translation of source code; special handling or control of the code, such as adding specialized debugging information; and control of the linker and library search paths. Options related to performance and optimization are summarized at the end of chapter "Getting started with optimization" on page 41. For description, full option syntax, and usage of each option, see *XL C/C++ for Mac OS X Compiler Reference*.

Basic translation

The options in this grouping have the broadest applicability for basic translation of source code. The subgroups of compiler options are generally concerned with:
- Standards compliance.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options related to basic translation of source code

Standards compliance	Compilation mode or control of compiler driver

`-qgenproto`, `-qnogenproto`
`-qlanglvl`
`-qlibansi`, `-qnolibansi`
	`-#`
	`-qaltivec, -qnoaltivec`
	`-F`
	`-qpath`
	`-qproto`, `-qnoproto`
	`-qsourcetype`

Source code generation	

`-qalloca`
`-qattr`, `-qnoattr`
`-B`
`-C`
`-qcpluscmt`, `-qnocpluscmt`
`-D`
`-qdbcs`, `-qnodbcs`
`-qdigraph`, `-qnodigraph`
`-E`
`-qignprag`
`-M`
`-ma`
`-qmacpstr`, `-qnomacpstr`
	`-qmakedep`
	`-qmbcs`, `-qnombcs`
	`-P`
	`-qrwvftable`, `-qnorwvftble`
	`-qsmallstack`, `-qnosmallstack`
	`-qsyntaxonly`
	`-t`
	`-qtabsize`
	`-qtrigraph`, `-qnotrigraph`
	`-U`
	`-qvrsave, -qnovrsave`
	`-W`

Diagnostics	**Compiled code**

Options related to basic translation of source code

-qflag -qinfo, -qnoinfo -qmaxerr, -qnomaxerr -qphsinfo, -qnophsinfo -qprint, -qnoprint -qshowinc, -qnoshowinc -qsource, -qnosource -qsrcmsg, -qnosrcmsg -qsuppress, -qnosuppress -V -v -w -qwarnfourcharconsts, -qnowarnfourcharconsts -qxcall, -qnoxcall	-qbitfields -c -qchars -qcommon, -qnocommon -qdollar, -qnodollar -o -qstatsym, -qnostatsym -qupconv, -qnoupconv

Special handling and control

The options in this grouping provide fine-grain control of the translation process and have less general applicability than basic translation options. The topics within this grouping of compiler options are generally concerned with:
- Data alignment.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options for special handling, fine tuning, and debugging

Data alignment	**Parallelization**
-qalign -qenum	-qthreaded, -qnothreaded
Floating-point and numerical features	
Sizes -qlonglong, -qnolonglong	***Rounding of floating-point values*** -y
Single-precision values None applicable for the Mac OS X platform.	***Other floating-point options*** -qfloat -qflttrap, -qnoflttrap
Debugging	
-qcheck, -qnocheck -qdbxextra, -qnodbxextra -qfullpath, -qnofullpath -g -qhalt	-qinitauto, -qnoinitauto -qlinedebug, -qnolinedebug -qlist, -qnolist -qlistopt, -qnolistopt -qsymtab -qxref, -qnoxref

Linking and library-related options

The options in this grouping are related to the linking phase of the compilation process. This grouping also contains options that provide specialized ways to specify search paths for finding libraries and header files. These compiler options are generally concerned with:
- Placing string literals and constants.
- Static and dynamic linking and libraries.
- Specifying search directories.

Options for controlling the ld command

Placing string literals and constants	Static and dynamic linking and libraires
-qkeyword, -qnokeyword -qro, -qnoro -qroconst, -qnoroconst	-qmkshrobj -qpic, -qnopic -qstdinc, -qnostdinc
Search directories	**Other linker options**
-I -L -l (lowercase el) -qc_stdinc -qcomplexgccincl, -qnocomplexgccincl -qframeworkdir -qgcc_c_stdinc -qidirfirst, -qnoidirfirst -r -qstdframework, -qnostdframework -Z	-bundle -bundle_loader -framework -qcrt, -qnocrt -qlib, -qnolib

Options summary: C++ compiler

Most of the C compiler options are available for compiling C++ programs. The following table presents additional compiler options specific to compiling C++ programs and the C options that are *not* available for compiling C++ programs on the Mac OS X platform:

Compiler options for C++ programs

C++-specific options	C-only options
-+ -qcinc -qcpp_stdinc -qeh, -qnoeh -qgcc_cpp_stdinc -qhaltonmsg -qpriority -qrtti, -qnortti -qstaticinline, -qnostaticinline -qtempinc, -qnotempinc -qtemplaterecompile, -qnotemplaterecompile -qtemplateregistry -qtempmax -qtmplparse -qvftable, -qnovftable	-qalloca -qassert, -qnoassert -qc_stdinc -qcpluscmt, -qnocpluscmt -qdbxextra, -qnodbxextra -qgcc_c_stdinc -qgenproto, -qnogenproto -ma -qproto, -qnoproto -qsrcmsg, -qnosrcmsg -qsyntaxonly -qupconv, -qnoupconv

Getting started with optimization

Simple compilation is a translation or transformation of the source code into an executable or shared object. An optimizing transformation is one that gives your application better overall performance at run time. XL C/C++ for Mac OS X provides a portfolio of optimizing transformations tailored to the PowerPC architecture. These transformations can:

- Reduce the number of instructions executed for critical operations.
- Restructure the generated object code to make optimal use of the PowerPC architecture.
- Improve the usage of the memory subsystem.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Their aim is to make your application run faster.

Significant performance improvements can be achieved with relatively little development effort if you understand the available controls that affect the transformation of well-written code. This section describes some of the optimizations the compiler can perform to help you balance the trade-offs among run-time performance, hand-coded micro-optimizations, general readability, and overall portability of your source code.

The optimization techniques discussed in this chapter were developed and are intended for applications that have not been vectorized. This discussion also assumes that you have used a profiler to identify the areas in your code where optimization might be appropriate.

Optimizations are often attempted in the later phases of application development cycles, such as product release builds. If possible, you should test and debug your code without optimization before attempting to optimize it. Embarking on optimization should mean that you have chosen the most efficient algorithms for your program and that you have implemented them correctly. To a large extent, compliance with language standards is directly related to the degree to which your code can be successfully optimized. Optimizers are the ultimate conformance test!

Optimization is controlled by compiler options, directives, and pragmas. However, compiler-friendly programming idioms can be as useful to performance as any of the options or directives. It is no longer necessary nor is it recommended to excessively hand-optimize your code (for example, manually unrolling loops). Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines. The section at end of this chapter called ″Compiler-Friendly Programming″ contains some suggested idioms and programming tips for writing good optimizable code.

It should be noted that not all optimizations are beneficial for all applications. A trade-off usually has to be made between an increase in compile time, accompanied by reduced debugging capability, and the degree of optimization done by the compiler.

Optimization levels

The default behavior of the compiler is to generate code without optimization and without debug information. Full debugging capability with the GNU debugger is supported, and enabled by compiling with the **-g** compiler option. The option **-g** can be used with optimization options, but with diminished debugging capability.

Optimization levels are specified by compiler options. A summary of the compiler behavior at each optimization level is shown in the following table.

Optimization levels

Option	Behavior
-qnoopt	Fast compilation, full debugging support.
-O2 (same as **-O**)	Comprehensive low-level optimization; partial debugging support.
-O3	More extensive optimization; some precision trade-offs.
-O4 and **-O5**	Interprocedural optimization; loop optimization; automatic machine tuning.

Optimizing for a particular processor architecture: target machine options

Target machine options are options that instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By default, the compiler generates code that runs on PowerPC 970-based Mac systems. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor. The following compiler options control optimizations affecting individual aspects of the target machine.

Target machine options

Option	Behavior
-qarch	Selects a family of processor architectures for which instruction code should be generated. This option restricts the instruction set generated to a subset of that for the PowerPC architecture.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of **-qtune**.

Selecting a predefined optimization level sets default values for these individual options.

Getting the most out of target machine options

Try to specify with **-qarch** the smallest family of machines possible that will be expected to run your code reasonably well.
- **-qarch=auto** generates code that may take advantage of instructions available only on the compiling machine (or on a system that supports the equivalent processor architecture).
- The default is **-qarch=ppcv**.

Try to specify with **-qtune** the machine where performance should be best. If you are not sure, try **-qtune=ppc970**, which is the default on Mac OS X.

The **-qarch** and **-qtune** options are closely related. The acceptable combinations of **-qarch** and **-qtune** are not problematic on the Mac OS X platform, unlike other platforms. The reason is that the only suboptions are **-qtune=auto** and the default,

-qtune=ppc970. The following table shows the architecture-related macros that are predefined by the compiler for each **-qarch** suboption.

-qarch	Predefined macro(s)	Default -qtune
ppcv	`_ARCH_PPCV`	ppc970
ppc970	`_ARCH_PPCV` `_ARCH_970` `_ARCH_G5`	ppc970
g5	`_ARCH_G5` `_ARCH_PPCV`	ppc970

Before using the **-qcache** option, look at the options sections of the listing using **-qlist** to see if the current settings are satisfactory. The settings appear in the listing itself when the **-qlistopt** option is specified.

If you decide to use **-qcache**, use **-qhot** or **-qsmp** along with it.

Optimization level -O2

At optimization level **-O2** (same as **-O**), the compiler performs comprehensive low-level optimization, which includes the following techniques.

- Global assignment of user variables to registers, also known as *graph coloring register allocation*.
- Strength reduction and effective use of addressing modes.
- Elimination of redundant instructions, also known as *common subexpression elimination*
- Elimination of instructions whose results are unused or that cannot be reached by a specified control flow, also known as *dead code elimination*.
- Value numbering (algebraic simplification).
- Movement of invariant code out of loops.
- Compile-time evaluation of constant expressions, also known as *constant propagation*.
- Control flow simplification.
- Instruction scheduling (reordering) for the target machine.
- Loop unrolling and software pipelining.

Partial debugging support at optimization level **-O2** consists of the following behaviors.

- Externals and parameter registers are visible at procedure boundaries, which are the entrance and exit to a procedure. You can look at them if you set a breakpoint at the entry to a procedure. However, function inlining with **-Q** can eliminate these boundaries and this visibility. This can also happen when the front end inlines very small functions.
- The `snapshot` pragma creates additional program points for storage visibility by flushing registers to memory. This allows you to view and modify the values of any local or global variable, or of any parameter in your program. You can set a breakpoint at the `snapshot` and look at that particular area of storage in a debugger.

Optimization level -O3

At optimization level **-O3**, the compiler performs more extensive optimization than at **-O2**. The optimizations may be broadened or deepened in the following ways.
- Deeper inner loop unrolling.
- Better loop scheduling.
- Increased optimization scope, typically to encompass a whole procedure.
- Specialized optimizations (those that might not help all programs).
- Optimizations that require large amounts of compile time or space.
- Implicit memory usage limits are eliminated (equivalent to compiling with **-qmaxmem=-1**).
- Implies **-qnostrict**, which allows some reordering of floating-point computations and potential exceptions.

Due to the implicit setting of **-qnostrict**, some precision trade-offs are made by the compiler, such as the following:
- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions (for example, division by zero, overflow).

Getting the most out of -O2 and -O3

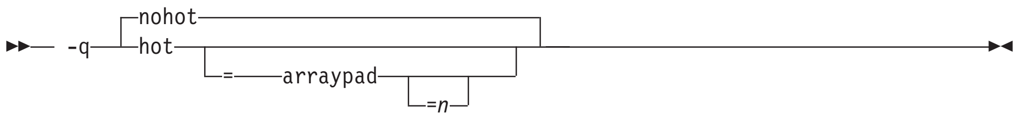
Here is a recommended approach to using optimization levels **-O2** and **-O3**.
- If possible, test and debug your code without optimization before using **-O2**.
- Ensure that your code complies with its language standard. Optimizers are the ultimate conformance test!

  In C code, ensure that the use of pointers follows the type restrictions: generic pointers should be **char*** or **void***. Also check that all shared variables and pointers to shared variables are marked **volatile**.
- Compile as much of your code as possible with **-O2**.
- If you encounter problems with **-O2**, consider using **-qalias=noansi** (C/C++) rather than turning off optimization.
- Next, use **-O3** on as much code as possible.
- If you encounter problems or performance degradations, consider using **-qstrict** or **-qcompact** along with **-O3** where necessary.
- If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1** or **-qnostrict**, or both.

High-order transformations (-qhot)

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling. The goals of these loop optimizations include:
- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements.

The supported syntax is as follows:



Compiling with **-qhot** turns on loop optimizations.

Getting the most out of -qhot

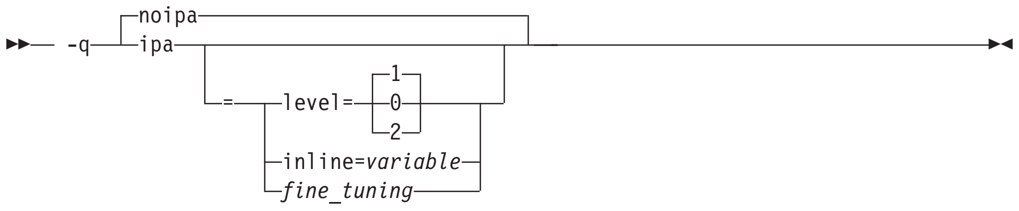
Try using **-qhot** along with **-O2** and **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.

- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of **-qhot**, try using **-qstrict** or **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Interprocedural analysis (-qipa)

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. Interprocedural analysis can be specified on the compile step only or on both compile and link steps ("whole program" mode). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object.

IPA is enabled by the option **-qipa**. The syntax for the most commonly used suboptions is shown in the simplified syntax diagram below. The full option syntax is described in the *XL C/C++ for Mac OS X Compiler Reference*.



The effects of these suboptions or groups of suboptions are described in the following table.

Commonly used -qipa suboptions

Suboption	Behavior
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partioning and layout of procedures according to their calling relationships, which is also referred to as their *call affinity*. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.
level=1	Inlining and global data mapping. Specifically, • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level when **-qipa** is specified.

Commonly used -qipa suboptions

Suboption	Behavior
level=2	Global alias analysis, specialization, interprocedural data flow. • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, code motion into conditions or out of loops, elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, and code motion across functions. • Procedure specialization (cloning).
inline=*variable*	Provides precise user control of inlining.
fine_tuning	Other values for **-qipa=** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Getting the most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions.

- When specifying optimization options in a makefile, remember to use the compiler driver (such as cc or xlc) to link, and to include all compiler options on the link step.
- **-qipa** works when building executables or shared objects, but always compile main and exported functions with **–qipa**.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- Ensure that there is enough space in /tmp (at least 200 MB), or use the TMPDIR environment variable to specify a different directory with sufficient free space.
- The **level** suboption is a throttle. Try varying it if link time is too long. Compiling with **–qipa=level=0** can be very beneficial for little additional link time.
- Look at the generated code after compiling with **-qlist** or **-qipa=list**. If too few or too many functions are inlined, consider using **–qipa=inline** or **–qipa=noinline**. To control inlining of a specific function, use **-Q+** and **-Q-**.
- If your application contains Fortran code that was compiled with IBM XL Fortran, you can also specify **-qipa** to compile that code using the XL Fortran compiler. Doing so produces additional optimization opportunities at link time. During linking, the **-qipa** option causes a complete reoptimization of the entire application.

The -O4 and -O5 macro options

Optimization levels 4 and 5 automatically activate several other optimization options. Optimization level 4 (**-O4**) includes:

- Everything from **-O3**
- **-qhot**
- **-qipa**
- **-qarch=auto**
- **-qtune=auto**
- **-qcache=auto**

Optimization level 5 (**-O5**) includes:

- Everything from **-O4**
- **-qipa=level=2**

Other program behavior options

The precision of compiler analyses is significantly affected by instructions that can read or write memory. Aliasing pertains to alternate names for things, which in this context are references to memory. A reference to memory can be direct, as in the case of a named symbol, or indirect, as in the case of a pointer or dummy argument. A function call might also reference memory indirectly. Apparent references to memory that are false, that is, that do not actually reference some location assumed by the compiler, constitute barriers to compiler analysis.

ISO C and C++ define a type-based aliasing rule. Simplified, the rule is that you cannot safely dereference a pointer that has been cast to a type that is not closely related to the type of what it points at. The language standards also define the closely related types.

Fortran defines a rule that dummy argument references may not overlap other dummy arguments or externally visible symbols during the execution of a subprogram.

The compiler performs sophisticated analyses, attempting to refine the set of possible aliases for pointer dereferences and calls. However, a limited scope and the absence of values at compile time constrain the effectiveness of these analyses. Increasing the optimization level, in particular, applying interprocedural analysis (that is, compiling with **-qipa**), can contribute to better aliasing.

Programs that violate language aliasing rules, as summarized above, commonly execute correctly without optimization or with low optimization levels, but can begin to fail when higher levels of optimization are attempted. The reason is that more aggressive optimizations take better advantage of aliasing information and can therefore expose subtly incorrect program semantics.

Options related to these issues are **-qstrict** and **-qalias**. Their behaviors are summarized in the table below.

Program behavior options

Option	Description
-qstrict, **-qnostrict**	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that may raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). The default is **-qstrict** with **-qnoopt** and **-O2**; **-qnostrict** with **-O3**, **-O4**, and **-O5**.
-qalias	Allows the compiler to assume that certain variables do not refer to overlapping storage. The focus is on overlap of storage accessed by pointers in C and C++ and on the overlap of dummy arguments and array assignments in Fortran. The full option syntax is described in *XL C/C++ for Mac OS X Compiler Reference*.

Diagnostic options

The following table presents options that provide specialized information, which can be helpful during the development of optimized code.

Diagnostic options

Option	Behavior
-qlist	Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions and text constants.

Diagnostic options

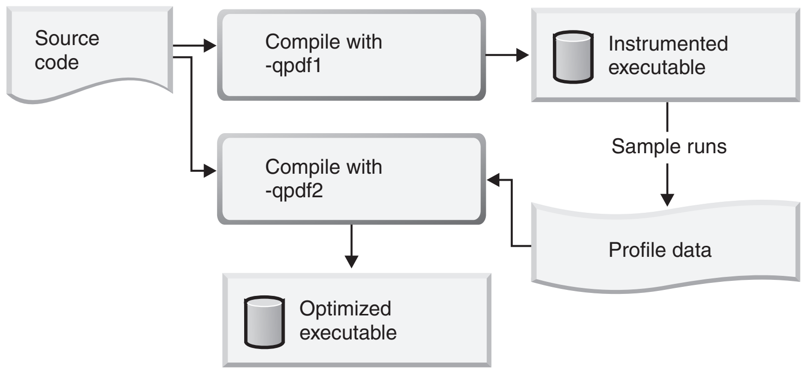
Option	Behavior
-qreport	Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. The option is enabled when **-qhot** or **-qsmp** is specified.
-qinitauto	Instructs the compiler to emit code that initializes all automatic variables to a given value.
-qcheck=nullptr, **-qcheck=bounds**, **-qcheck=divzero**	Inserts run-time checks (trap instructions) for null pointer access, array bounds violations, or division by zero.
-qipa=list	Instructs the compiler to emit an object listing.

Profile-directed feedback (PDF)

Profile-directed feedback is a two-stage compilation process that lets you provide the compiler with data characteristic of typical program behavior. An instrumented executable is run in a number of different scenarios for an arbitrary amount of time, producing a profile data file as a side effect. A second compilation using the profile data produces an optimized executable.

PDF should be used mainly on code that has rarely executed conditional error handling or instrumentation. The technique has a neutral effect in the absence of firm profile information, but is not recommended if insufficient or uncharacteristic data is all that is available.

The following diagram illustrates the PDF process. Not all the code in an application needs to be compiled and linked with **-qpdf1** or **-qpdf2** to benefit from this process.



The two stages of the process are controlled by the compiler options **-qpdf1** and **-qpdf2**. Stage 1 is a regular compilation using an arbitrary set of optimization options and **-qpdf1**, that produces an executable or shared object that can be run in a number of different scenarios for an arbitrary amount of time. Stage 2 is a recompilation using the same options, except **-qpdf2** is used instead of **-qpdf1**, during which the compiler consumes previously collected data for the purpose of path-biased optimization.

If your application contains Fortran code compiled with IBM XL Fortran, you can achieve additional PDF optimization by specifying the **-qpdf1** and **-qpdf2** options, which are also available on the Fortran compiler. If you combine **-qpdf1** or **-qpdf2** with **-qipa** or the **-O5** option (that is, if you link with IPA) on all Fortran and C or C++ code, you will maximize the amount of PDF information that is available for optimization.

Other performance options

Options are provided to control particular aspects of optimization. They are often enabled as a group or given default values when a more general optimization option is enabled.

Selected compiler options for optimizing performance

Option	Description
-qcompact	Chooses reduction of final code size over a reduction in execution time when a choice is necessary. Can be used to constrain **-O3** optimization.
-qsmallstack	Instructs the compiler to compact stack storage. Doing so may increase heap usage.
-qinline	Controls inlining of named functions. Can be used at compile time, link time, or both. When **-qipa** is used, **-qinline** is synonymous with **-qipa=inline**.
-qunroll	Independently controls loop unrolling. Is implicitly activated under **-O3**.
C++ **-qeh**	Enables full support for C++ exceptions.
C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted.
-qunwind	Informs the compiler that the stack can be unwound while a routine in this compilation is active. In other words, the compiler is informed that the application does not rely on any program stack unwinding mechanism.
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. For C++, this option implies **-qnoeh**: if **-qnounwind** is enabled and an exception is thrown, the program might crash. The option **-qnounwind** enables optimization prologue tailoring, which reduces the number of saves and restores of nonvolatile registers.

Floating-point options

Special compiler options exist for handling floating-point calculations efficiently. By default, the compiler makes a trade-off to violate certain IEEE 754 floating-point rules in order to improve performance. For example, multiply-add instructions are generated by default because they are faster and produce a more precise result than separate multiply and add instructions. Floating-point exceptions, such as overflow or division by zero, are masked by default. If you need to catch these exceptions, you have the choice of enabling hardware trapping of these exceptions or using software-based checking. The option **-qflttrap** enables software-based checking.

Options for handling floating-point calcluations

Option	Description
-qfloat	Provides precise control over the handling of floating-point calculations.
-qflttrap	Enables software checking of IEEE floating-point exceptions. This technique is sometimes more efficient than hardware checking because checks can be executed less frequently.

See *XL C/C++ for Mac OS X Compiler Reference* for more information.

Compiler-friendly programming

Compiler-friendly programming idioms can be as useful to performance as any of the options or directives. Here are some suggestions.

General
- Use the xlc or xlc_r invocation rather than cc or cc_r, when possible.
- Always include `string.h` when doing string operations and `math.h` when using the math library.

Hand-tuning
- Do not excessively hand-optimize your code. Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines.
- Do limited hand tuning of small functions by defining them as **inline** in a header file.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using **–qipa**.

Variables
- Avoid unnecessary use of global variables and pointers. When using them in a loop, load them into a local variable before the loop and store them back after.
- Use **volatile** only for truly shared variables.
- Use **const** for globals, parameters, and functions wherever possible.

Conserving storage
- Use register-sized integers (**long** data type) for scalars. For large arrays of integers, consider using one- or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.
- Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.

Pointers
- Obey all language aliasing rules. Try to avoid using **–qalias=noansi**.
- Use unions and pointer type casting only when necessary.
- Pass large class or struct parameters by address or reference; pass everything else by value where possible.

Arrays
- Use local variables wherever possible for loop index variables and bounds. Avoid taking the address of loop indices and bounds.
- Keep array index expressions as simple as possible.

See *XL C/C++ for Mac OS X Programming Tasks* for more information.

Options summary: optimization and performance

This chapter appendix presents a summary of the C compiler options that deal with optimization and performance tuning. The options are grouped by type. Other C compiler options are summarized at the end of the chapter “Getting started with compiler options” on page 29. For description, full option syntax, and usage, see *XL C/C++ for Mac OS X Compiler Reference*.

Options related to optimization and performance tuning

Optimization flags	Restricting optimization options

Options related to optimization and performance tuning

-O -qoptimize -qagrrcopy	-qstrict, -qnostrict
Aliasing	**Inlining functions**
-qalias -qansialias, -qnoansialias -qassert, -qnoassert	-Q -qinline, -qnoinline
Side effects	**Code size reduction**
-qignerrno, -qnoignerro -qisolated_call	-qcompact, -qnocompact -s
Compile-time optimization	**Loop optimization**
-qmaxmem -qspill -qunwind, -qnounwind	-qhot, -qnohot -qreport, -qnoreport -qstrict_induction, -qnostrict_induction -qunroll, -qnounroll
Processor and architectural optimization	**Whole-program analysis**
-qarch -qcache -qtune	-qipa, -qnoipa
Performance data collection	**Other optimization options**
-p -qpdf1, -qnopdf1 -qpdf2, -qnopdf2 -pg	None applicable for the Mac OS X platform.

Porting considerations

Porting an existing AIX or other UNIX®-based application to the Mac OS X platform can involve more than one area of investigation. You can filter the diagnostic messages emitted by the compiler to show only those that pertain to portability issues. Compiling with the **-qinfo=por** option enables this filter.

Some areas related to porting to the Mac OS X platform are:

- Checking the amount of reliance on GNU C and other language extensions. An application that conforms strictly to its ISO language specification will be maximally portable. Currently, IBM XL C/C++ for Mac OS X supports a subset of the GNU C and C++ extensions to C and C++. You may need to revisit code that relies on unsupported extensions.
- Checking how null pointers are dereferenced. Some errors in the code can go undetected on a platform due to operating system-dependent characteristics. This kind of error might show up when the program is ported to another platform. Use the option **-qcheck=nullptr** to help detect such conditions before porting.

  The lowest 4K of memory (that is, addresses 0 through 4K-1) are readable and contain zeroes on AIX, but are not readable on the Linux and Mac OS X platforms, and will cause a segmentation violation if accessed. For example,

  ```
  if (strcmp(a, NULL) == 0) ...
  ```

  results in a segmentation violation on Linux and Mac OS X, but not on AIX.
- Checking the alignment. The supported types of alignment for AIX, Linux, and Mac OS X are not all the same. If you are porting a program that relies on specific values for **-qalign** or `#pragma align`, you may need to change the program. This is especially true if you use or plan to use AltiVec vector types and programming constructs, which impose additional constraints on alignment.
- Ensuring the portability of data structures. If you generate data with an application on one platform and read the data with an application on another platform, the data may have an alignment that is different from that which the reading application expects. To avoid this problem, make sure that you use a platform-neutral mechanism for the layout of data in structures. For example, if you enclose a structure with `#pragma pack(1)` and `#pragma pack(pop)` pair, the alignment will be the same on all platforms.
- Using the **gxlc** or **gxlc++** utility for translating the commands in your makefiles.
- On Linux or Mac OS X, if the default global `operator new` is called and the allocation request cannot be fulfilled, an exception of type `std::bad_alloc` is thrown. On AIX, the default behavior of the global `operator new` is to return a null pointer if allocation fails.
- Ensuring the portability of applications that use templates. The C++ compiler provides two different methods of working with template files. Each method has an associated compiler option. The `-qtemplateregistry` compiler option maintains a record of all templates. This method is recommended. An older compiler option, `-qtempinc`, is also provided for applications that you port from another platform. However, on the Mac OS X platform, the compiler option `-qtempinc` is considered deprecated.

Features related to GNU C and C++ portability

To ease porting an application or code developed with GNU C, XL C/C++ for Mac OS X supports a subset of the GNU C and C++ language extensions. The tables in this section list the extensions features that are supported and those for which the syntax is accepted but the semantics ignored.

To use *supported* extensions with your C code, use the **xlc** or **cc** invocation commands, or specify one of `-qlanglvl=extc89`, `-qlanglvl=extc99`, or `-qlanglvl=extended`. To use these features with your C++ code, specify the `-qlanglvl=extended` option. In C++, all supported GNU C and C++ features are accepted by default.

The compiler recognizes *accept/ignore* extensions as acceptable programming keywords, but does not support them.

Compiling source code that uses these extensions under a strict language level (`stdc89`, `stdc99`) will result in error messages.

The GNU C and C++ language extensions are fully documented in the GNU manuals. See http://gcc.gnu.org/onlinedocs.

GCC function attributes

Use the keyword **__attribute__** to specify special attributes when making a function declaration. This keyword is followed by an attribute specification inside double parentheses.

GCC function attribute compatibility with XL C/C++ for Mac OS X

Function Attribute	Behavior
`alias`	unsupported
`cdecl`	accept/ignore
`const`	supported
`constructor`	unsupported
`destructor`	unsupported
`dllexport`	accept/ignore
`dllimport`	accept/ignore
`eightbit_data`	accept/ignore
`exception`	accept/ignore
`format`	accept/ignore
`format_arg`	accept/ignore
`function_vector`	accept/ignore
`interrupt`	accept/ignore
`interrupt_handler`	accept/ignore
`longcall`	accept/ignore
`model`	accept/ignore
`no_check_memory_usage`	accept/ignore
`no_instrument_function`	accept/ignore
`noreturn`	supported

GCC function attribute compatibility with XL C/C++ for Mac OS X

Function Attribute	**Behavior**
`pure`	supported
`regparm`	accept/ignore
`section`	unsupported
`stdcall`	accept/ignore
`tiny_data`	accept/ignore
`weak`	unsupported

Related References

- Function Attributes in *XL C/C++ for Mac OS X C/C++ Language Reference*

GCC variable attributes

Use the keyword **__attribute__** to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses.

GCC variable attribute compatibility with XL C/C++ for Mac OS X

Variable Attribute	**Behavior**
`aligned`	supported
▶ C++ `init_priority`	supported
`mode`	supported
`model`	accept/ignore
`nocommon`	supported
`packed`	supported
`section`	unsupported
`transparent_union`	accept/ignore
`unused`	accept/ignore
`weak`	unsupported

Related References

- Variable Attributes in *XL C/C++ for Mac OS X C/C++ Language Reference*

GNU C and C++ type attributes

Use the keyword **__attribute__** to specify special attributes of `struct` and `union` types when you define these types. This keyword is followed by an attribute specification inside double parentheses. While type attributes are not currently supported in XL C/C++ for Mac OS X, some type attributes are accepted and ignored by the compiler.

GCC type attribute compatibility with XL C/C++ for Mac OS X

Type Attribute	**Behavior**
`aligned`	accept/ignore
`packed`	accept/ignore
`transparent_union`	accept/ignore

GCC type attribute compatibility with XL C/C++ for Mac OS X

Type Attribute	Behavior
`unused`	accept/ignore

GNU C and C++ assertions

Use *assertions* to test what sort of computer or system the compiled program will run on. The assertions `#cpu`, `#machine`, and `#system` are predefined. You can also define assertions with the preprocessing directives `#assert` and `#unassert`.

GNU C and C++ assertions in XL C/C++ for Mac OS X

GNU C Assertions	Behavior
`#assert`	supported
`#unassert`	supported
`#cpu`	supported possible value is `powerpc`
`#machine`	supported possible values are `powerpc` and `bigendian`
`#system`	supported possible value are `unix` and `posix`

Related References
- "Language support," on page 57

Other extensions related to GNU C and C++

The following features related to GNU C and C++ are supported under extended language levels (`extc89`, `extc99`, `extended`).

- Use directive `#warning` to cause the preprocessor to issue a warning and continue processing.
- Use directive `#include_next` to specify inclusion of the next header file in a directory after the current one.
- Local labels can be declared at the start of each statement expression.
- Use the address of a label as a constant of type `void*`.
- Use a brace-enclosed compound statement inside of parentheses as an expression.
- Refer to the type of an expression with the **__typeof__** keyword.
- Use compound expressions, conditional expressions, and casts as lvalues.
- Use keyword **__alignof__** to inquire about variable alignment, or the alignment usually required by a type.
- Use alternate spelling of these keywords: **__const__**, **__volatile__**, **__signed__**, **__inline__**, and **__typeof__**.

Under extended language levels (`extc89`, `extc99`, `extended`), XL C/C++ for Mac OS X recognizes the syntax of the following features, but their sematics are not supported.

- The declaration of a register variable, either global or local, can suggest a preferred register.

Many existing extensions to IBM C/C++ are also supported in GNU C and C++.

Appendix. Language support

Syntax and semantics constitute a complete specification of a programming language, but complete implementations can differ due to extensions. Pragmatic considerations, advances in programming techniques, and the shifting needs of modern programming environments are factors that influence growth and change.

XL C/C++ can foster a programming style that emphasizes portability. A program that conforms strictly to its language specification will have maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler will compile and execute properly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the language that are provided by the language implementation can improve the efficiency of its object code.

ISO/IEC 14882:1998 International Standard compatibility

The ISO/IEC 14882:1998 International Standard (also known as Standard C++) specifies the form and establishes the interpretation of programs written in the C++ programming language. This International Standard is designed to promote the portability of C++ programs among a variety of implementations. For strict conformance to Standard C++, use the `-qlanglvl=strict98` compiler option.

ISO/IEC 14882:1998 is the first formal definition of the C++ language.

ISO/IEC 9899:1990 International Standard compatibility

The ISO/IEC 9899:1990 International Standard (also known as C89) specifies the form and establishes the interpretation of programs written in the C programming language. This specification is designed to promote the portability of C programs among a variety of implementations. This Standard was amended and corrected by ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. To ensure that your source code adheres strictly to the amended and corrected C89 standard, specify the `-qlanglvl=stdc89` compiler option.

ISO/IEC 9899:1999 International Standard support

The ISO/IEC 9899:1999 International Standard (also known as C99) is an updated standard for programs written in the C programming language. It is designed to enhance the capability of the C language, provide clarifications to C89, and incorporate technical corrections. XL C/C++ for Mac OS X supports many features of this language specification.

▶ C The C compiler supports all language features specified in the C99 Standard. To ensure that your source code adheres to this set of language features, use the **`c99`** invocation command. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment. The availability of system header files provides an indication of whether such support exists.

Major features in C99

XL C/C++ for Mac OS X implements all C99 language features. The following is a table of selected major features.

ISO/IEC 9899:1999 international standard extensions to IBM C

C99 Feature	**Related Reference**
`restrict` type qualifier for pointers	The `restrict` Type Qualifier in *XL C/C++ for Mac OS X C/C++ Language Reference*
universal character names	The Unicode Standard in *XL C/C++ for Mac OS X C/C++ Language Reference*
predefined identifier `__func__`	Predefined Identifiers in *XL C/C++ for Mac OS X C/C++ Language Reference*
function-like macros with variable and empty arguments	Function-Like Macros in *XL C/C++ for Mac OS X C/C++ Language Reference*
`_Pragma` unary operator	The _Pragma Operator in *XL C/C++ for Mac OS X C/C++ Language Reference*
variable length array	Arrays in *XL C/C++ for Mac OS X C/C++ Language Reference*
`static` keyword in array index declaration	Arrays in *XL C/C++ for Mac OS X C/C++ Language Reference*
`complex` data type	Complex Types in *XL C/C++ for Mac OS X C/C++ Language Reference*
`long long int` and `unsigned long long int` types	Integer Variables in *XL C/C++ for Mac OS X C/C++ Language Reference*
hexadecimal floating-point constants	Hexadecimal Floating Constants in *XL C/C++ for Mac OS X C/C++ Language Reference*
compound literals for aggregate types	Compound Literals in *XL C/C++ for Mac OS X C/C++ Language Reference*
designated initializers	Initializers in *XL C/C++ for Mac OS X C/C++ Language Reference*
C++ style comments	Comments in *XL C/C++ for Mac OS X C/C++ Language Reference*
implicit function declaration not permitted	Function Declarations in *XL C/C++ for Mac OS X C/C++ Language Reference*
mixed declarations and code	for Statement in *XL C/C++ for Mac OS X C/C++ Language Reference*
`_Bool` type	Simple Type Specifiers in *XL C/C++ for Mac OS X C/C++ Language Reference*
`inline` function declarations	Inline Functions in *XL C/C++ for Mac OS X C/C++ Language Reference*
initializers for aggregates	Initializing Arrays Using Designated Initializers in *XL C/C++ for Mac OS X C/C++ Language Reference*

Changes and clarifications of C89 supported in C99

Certain specifications in the C99 standard are based on changes and clarifications of the C89 standard, rather than on new features of the language. XL C/C++ for Mac OS X supports all C99 language features, including the following:

- Flexible array members are allowed. The last member of a structure with two or more members can be declared without the size.
- Declaring implicit `int` is not supported. All declarations must have a type specifier.
- Trailing commas are allowed in enumeration specifiers.
- Duplicate type qualifiers are accepted and ignored, unless explicitly specified otherwise.
- A diagnostic message will be issued if a required expression is missing from the return statement.
- Constant expressions evaluated during preprocessing now use `long long` and `unsigned long long` data types.
- Empty macro arguments are allowed in function-like macros.
- The maximum value of `#line` has increased to 2 147 483 647.

C99 features in XL C/C++

Some features of the ISO/IEC 9899:1999 International Standard (C99) are also implemented in C++. These extensions are available under the `-qlanglvl=extended` compiler option.

ISO/IEC 9899:1999 international standard extensions to IBM C++

C99 Feature	Reference
`restrict` type qualifier for pointers	The `restrict` Type Qualifier in *XL C/C++ for Mac OS X C/C++ Language Reference*
predefined identifier `__func__`	Predefined Identifiers in *XL C/C++ for Mac OS X C/C++ Language Reference*
function-like macros with variable and empty arguments	Function-Like Macros in *XL C/C++ for Mac OS X C/C++ Language Reference*
`_Pragma` unary operator	The _Pragma Operator in *XL C/C++ for Mac OS X C/C++ Language Reference*

Enhanced language level support

The `-qlanglvl` compiler option is used to specify the supported language level, and therefore affects the way your code is compiled. You can also specify the language level implicitly by using different compiler invocation commands. In general, a valid program that compiles and runs correctly under a standard language level should continue to compile correctly and run to produce the same result with the orthogonal extensions enabled.

For example, to compile C programs so that they comply strictly with the ISO/IEC 9899:1990 International Standard (C89), you need to specify `-qlanglvl=stdc89`. The `stdc89` suboption instructs the compiler to strictly enforce the standard, and not to allow any language extensions. (The **c89** compiler invocation command specifies this language level implicitly.)

To compile C++ programs so that they conform strictly to ISO/IEC 14882:1998 International Standard (Standard C++), specify `-qlanglvl=strict98`.

You can also use extensions to the standard language levels. Extensions that do not interfere with the standard features are called *orthogonal* extensions. For example, when you compile C programs, you can enable extensions that are orthogonal to C89 by specifying `-qlanglvl=extc89`.

Most of the language features described in the ISO/IEC 9899:1999 International Standard (C99) are considered orthogonal extensions to C89.

When you compile C++ programs, you can enable the use of orthogonal extensions by specifying `-qlanglvl=extended`.

Non-orthogonal extensions, on the other hand, can interfere or conflict with aspects of the language as described in one of the international standards. Acceptance of these extensions must be explicitly enabled by a particular compiler option. For example, to support AltiVec vector types and programming constructs, the C compiler requires non-orthogonal extensions to the language, which are enabled by the option `-qaltivec`. Reliance on non-orthogonal extensions reduces the ease with which your application can be ported to different environments.

The main suboptions for the `-qlanglvl` option are listed below.

Selected -qlanglvl suboptions

-qlanglvl Suboption	**Suboption Description**
`-qlanglvl=stdc99`	C Specifies strict conformance to the C99 standard.
`-qlanglvl=stdc89`	C Specifies strict conformance to the C89 standard.
`-qlanglvl=strict98`	C++ Specifies strict conformance to Standard C++. Identical to `-qlanglvl=ansi`.
`-qlanglvl=extc99`	C Enables all extensions orthogonal to C99.
`-qlanglvl=extc89`	C Enables all extensions orthogonal to C89.
`-qlanglvl=extended`	C Enables all extensions orthogonal to C89 and specifies the `-qupconv` compiler option. C++ Enables all the orthogonal extensions on top of Standard C++.

Related References

- langlvl in *XL C/C++ for Mac OS X Compiler Reference*
- upconv in *XL C/C++ for Mac OS X Compiler Reference*
- longlong in *XL C/C++ for Mac OS X Compiler Reference*
- The IBM Language Extensions in *XL C/C++ for Mac OS X C/C++ Language Reference*
- The IBM C Language Extensions in *XL C/C++ for Mac OS X C/C++ Language Reference*
- The IBM C++ Language Extensions in *XL C/C++ for Mac OS X C/C++ Language Reference*

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtains the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	POWER	pSeries
IBM	PowerPC	VisualAge

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names, may be trademarks or service marks of others.

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899–1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).

Program Number: 5724–G12

SC09-7859-00